Exact approaches for solving a covering problem with capacitated subtrees

François Clautiaux, Jeremy Guillot, Pierre Pesneau

Exact approaches for solving a covering problem with capacitated subtrees

Clautiaux, François[a,b], Guillot, Jérémy[a,b], Pesneau, Pierre[a,b,*]

[a] *Université de Bordeaux, Institut de Mathématiques de Bordeaux UMR 5251, 351, cours de la Libération, 33405 Talence, France.*
[b] *Inria Bordeaux - Sud-Ouest, 200, avenue de la Vieille Tour, 33405 Talence, France.*

Abstract

In this document, we present a covering problem where vertices of a graph have to be covered by rooted subtrees. We present three mixed-integer linear programming models, two of which are compact while the other is based on Dantzig-Wolfe decomposition. In the latter case, we focus on the column generation subproblem, for which we propose several algorithms. Numerical results are obtained using instances from the literature and instances based on a real-life districting application. Experiments show that the branch-and-price algorithm is able to solve much bigger instances than the compact model, which is limited to very small instance sizes.

Keywords: Covering problems, Column generation, Dantzig-Wolfe decomposition, Branch-and-bound, Dynamic programming

Declarations of interest none.

Potential referees .

Black and white print this draft can be printed in black and white.

1. Introduction

In this document, we present a covering problem that arises in the application field of solid waste collection. Our problem belongs to the family of *districting problems*. The term "districting" refers to the operation of regrouping small geographical areas, called *basic units*, in bigger groups, called *districts*. In this work, basic units are presented as vertices of a graph, and districts are rooted subtrees.

For fifty years, researchers have been studying this family of problems for political districting [1, 2], sales territory design [3, 4, 5] and the optimization of public services [6, 7] (see, for example, Kalcsics' recent literature survey [8]). In any districting problem, districts are required to be *"connected"*, *"compact"* and *"balanced"*. An important issue in this field of research is to formally define these three concepts, whose meaning strongly depends on the industry they apply to.

The connectivity constraint implies that any pair of vertices belonging to the same district can be connected by some path whose vertices also belong to that district. This constraint is usually meant to prevent intersecting districts, but it can also be used to represent geographical obstacles. In [4], the authors used the original connectivity constraint, which leads to linear programs with an exponential number of constraints. In [3] and [2], the authors proposed a more restrictive version of connectivity, which uses the notion of *center*. For each district, one must set a center j; when the problem is defined on a graph, the center corresponds to a vertex. The connectivity constraint is verified if every district is a subtree of a spanning tree rooted in j. In [3] and [2], the authors used a shortest path tree to define the spanning tree, and thus the connectivity constraint. The use of the shortest path tree is motivated in [2] by the fact that a connected and compact sector is likely to contain the shortest path from the center to every node it contains. We use the same approach in this document.

Out of the three concepts (connectivity, compactness, balance), compactness is the one that is most open to interpretation. It is generally part of the objective function. Compactness can be modeled in a p-median problem fashion [3, 2], where one minimizes the sum of the distances between any vertex and the center of the district it belongs to, or in a p-center problem fashion [4, 5], where one only takes into account the maximum distance, among all districts, between a vertex and its corresponding center. Our approach would be closer to the p-center model since we consider, for each district, the maximum shortest path distance between a vertex and its center. In this work, that distance is called the *radius* of the district. However, we define the total cost of a solution as the sum of the district radiuses instead of the maximum district radius. In this regard, our model can be compared to that of [7] and [9]. One of the criteria of [7] was the sum of the *nor-*

*Corresponding author
Email addresses: francois.claurtiaux@math.u-bordeaux.fr (Clautiaux, François), jeremy.guillot@math.u-bordeaux.fr (Guillot, Jérémy), pierre.pesneau@math.u-bordeaux.fr (Pesneau, Pierre)

malized radiuses for all districts, while [9] used the sum of the district diameters, *i.e.* the maximum distance between any two vertices belonging to the same district.

The balance constraint implies that the difference of volume between any two districts must be small, with regards to workload, service time or population size, for example. In most cases, this constraint requires, for each attribute being considered, that the volume of any district is close to the mean, by a given margin. This constraint can also be represented by an objective function minimizing the distance between each district and the mean [9].

In our problem, districts are not subject to a minimum workload. In the real-life instances we studied, balance was not an industrial requirement. However, workload is bounded from above, which prevents the districts from having an excessive workload. The number of districts to compute is given, and each district is subject to two capacity constraints. Another specificity of our problem is that it consists in finding a vertex cover in the graph instead of a partition. In cases where some vertices belong to many shortest paths, it helps ensuring that a feasible solution is computed.

Several papers on districting problems describe heuristics based on local search [6, 4, 9]. There are works based on mathematical programming as well, and more specifically Dantzig-Wolfe decomposition [2]. Column generation methods are well-suited for this family of problems, since they can easily be decomposed in one partitioning/covering master problem and a subproblem for each district.

The aim of our work is to solve exactly this problem using algorithms based on mathematical programming. In particular, we focus on a branch-and-price method. The main difficulty is to design an efficient subproblem oracle which, for a given center, computes the subtree rooted in this center that minimizes the reduced cost. This subproblem can also be seen as an extension of the knapsack problem with precedence constraints, two capacity constraints, and a profit that depends on the maximum distance from the center (radius). We propose several methods for solving this problem. The first method is based on executing, for all relevant radius values, a two-dimensional generalization of the branch-and-bound algorithm proposed in [10]. The second method solves a dynamic program, based on the algorithm proposed in [11], that includes the radius in its recurrence relation.

We validate our methods numerically using real-life instances and instances from the vehicle routing literature. Our experiments show that the model based on column generation exhibits much better performances than the compact models. Moreover, the enumerative branch-and-bound oracle seems to be the best method to solve the column generation subproblems, although the dynamic programming algorithm can be faster in a significant number of cases.

In section 2, we present the problem formally, along with the different models studied in this document. We dedicate section 3 to the column generation subproblem while section 4 describes the techniques we use in the branch-and-price algorithm. We analyze the numerical results in section 5 before offering some brief concluding remarks and suggestions for future research.

2. Problem definition and mathematical models

In this section, we formally define the problem we are studying. For this purpose, we disregard the industrial application that motivates this work and focus on a vertex covering problem restricted to subtrees.

2.1. Formal problem definition

We represent the geographical area to be serviced using a weighted directed graph $G = (V, A, c)$ in which each vertex $i \in V$ represents a waste collection point and each arc $(i, j) \in A$ represents a connection from i to j of length $c_{i,j}$. Let $n = |V|$. Any vertex $i \in V$ is also associated with two values $w_i^1 \in \mathbb{N}$ and $w_i^2 \in \mathbb{N}$ corresponding, respectively, to the time required to service that point and the volume of waste to be collected therein. The fleet is composed of K vehicles, each of them limited to a maximum total service time $W^1 \in \mathbb{N}$ and a capacity $W^2 \in \mathbb{N}$ of waste that can be collected.

Each *district* that we want to generate is defined by a pair (U, j), where $U \subseteq V$ is a set of vertices, and $j \in U$ is the district's *center*. For a given district centered at j, let $d_{i,j}$ be the distance of the shortest path in G from j to vertex i and let $\pi_j(i)$ be the vertex preceding i in this path. If there is no path from j to i in G, then $d_{i,j} = +\infty$ and we assume, by convention, that $\pi_j(i) = j$. These shortest paths define a tree $T^j = (V, A^j)$ rooted in j with $A^j = \{(\pi_j(i), i), i \in V \setminus \{j\}\}$. The *radius* of a district (U, j) is defined by $\max_{i\in U} d_{i,j}$. In addition, for a district (U, j), we have that $i \in U \implies \pi_j(i) \in U$.

In the following, given $T^j = (V, A^j)$ a tree rooted in $j \in V$ and $i \in V \setminus \{j\}$, we define $T^j(i)$ the subtree of T^j rooted in i and composed of all the descendants of i. In addition, given a set of vertices $U \subseteq V$, we define $T^j[U] = (U, A^j \cap (U \times U))$ the subtree of T^j rooted in j and containing every vertex of U. We say that set U induces a valid subtree of T^j if $j \in U$ and $T^j[U]$ is a connected component of T^j.

Definition 1 (valid district, radius). *Given a tree $T^j = (V, A^j)$ rooted in j, three vectors $\boldsymbol{d}_{\cdot,j}$, $\boldsymbol{w}^1$, $\boldsymbol{w}^2$ in $\mathbb{R}^n_+$, and two real numbers W^1 and W^2, a* valid district centered at *j is a pair (U, j) such that $T^j[U]$ is a valid subtree and $\sum_{i\in U} w_i^\ell \leq W^\ell$ for $\ell = 1, 2$. The* radius *of a valid district (U, j) is defined as $r(U, j) = \max_{i\in U} d_{i,j}$.*

Problem 1 (minimum cost cover with valid districts)**.** *Given V a set of vertices, two vectors $\boldsymbol{w^1}$ and $\boldsymbol{w^2}$ in $\mathbb{N}^n$, two integer values $W^1 \geq 1$ and $W^2 \geq 1$, an integer K, and for each vertex $j \in V$, a tree T^j of shortest paths from j to any other vertex in V, and a vector $\boldsymbol{d}_{\cdot,j} \in \mathbb{R}^n$*

of shortest path distances from j to any other vertex in V, find a cover of V using K valid districts with distinct centers and minimizing the sum of the districts' radiuses.

The problem of covering a graph with valid districts is NP-hard, since the bin packing problem is reducible to this covering problem with trees of depth 1, null distances and capacity W^2 equal to $+\infty$.

Note that an item may belong to two different districts (and thus counts in two knapsack constraints) in a solution, which may be counter-intuitive. However, this is mandatory in order to keep the precedence constraints relevant. It also helps finding solutions in which few items are covered more than once.

2.2. Compact formulations

The problem of vertex covering using valid districts of minimum cost can be formulated as an integer linear program. Our first model considers natural decision variables (center selection, assignments, and radiuses). Let $\boldsymbol{x} \in \{0,1\}^{n\times n}$ be a matrix of decision variables. For $i \neq j$, $x_{i,j}$ is equal to 1 if i is assigned to center j, 0 otherwise. Each variable $x_{j,j}$ is equal to 1 if j is the center of some district, and 0 otherwise. Let also $\boldsymbol{r} \in \mathbb{R}^n$ be a vector of variables. For each vertex $j \in V$, r^j is equal to the radius of the district centered at j if j is a center, and 0 otherwise. The problem can then be modeled as follows:

$$
(\mathrm{P}_0) = \begin{cases}
\min \sum_{j\in V} r^j & & & \\
\text{s.t.} \sum_{j\in V} x_{j,j} \leq K, & & & (1)\\
\sum_{j\in V} x_{i,j} \geq 1 & \forall i \in V, & & (2)\\
\sum_{i\in V} w_i^\ell x_{i,j} \leq W^\ell & \ell = 1,2, & \forall j \in V, & (3)\\
x_{i,j} \leq x_{\pi_j(i),j} & \forall i \in V, i\neq j, & \forall j \in V, & (4)\\
r^j \geq d_{i,j}\, x_{i,j} & \forall i \in V, & \forall j \in V, & (5)\\
r^j \in \mathbb{R}^+ & & \forall j \in V, & (6)\\
x_{i,j} \in \{0,1\} & \forall i \in V, & \forall j \in V. & (7)
\end{cases}
$$

The objective function is equal to the sum of the radiuses. Note that although this is not explicitly enforced, when j is not a center, $r^j = 0$ in any optimal solution. Constraint (1) is an upper bound for the number of districts used ; constraints (2) ensure that each vertex belongs to at least one district ; and constraints (3) guarantee that no district can exceed any of the two maximum capacities. Each constraint (4) asks that if i is assigned to cluster j, then its predecessor is assigned to j as well. Constraints (5) guarantee that each variable r^j carries the corresponding radius value.

We now propose a better model for the problem, with different variables. For that purpose, we use a model similar to that of [12] for the p-center problem. For each vertex $j \in V$, let σ_j be a permutation of the vertices in V in order of increasing distance from center j. For a given $j \in V$ and $i \in \{1,\dots,n\}\setminus\{j\}$, let $\Delta_{i,j}$ be the difference between the distances (from center j) of vertex i and its predecessor in σ_j. It can be defined as follows: $\Delta_{j,j} = 0$ and $\Delta_{\sigma_j(k),j} = d_{\sigma_j(k),j} - d_{\sigma_j(k-1),j}$ for $k = 2,\dots,n$. We can now distribute the maximum distance in a district among all the vertices it contains. To this end, we introduce a new matrix $\boldsymbol{y} \in \{0,1\}^{n\times n}$ of binary variables. For any valid pair (i,j), we have that $y_{i,j}$ is equal to 1 if j is a center and the maximum distance in district j is greater than or equal to $d_{i,j}$, and 0 otherwise. Thus, if j is chosen as a center, and vertex $\sigma_j(i)$ defines the radius of district j, we have $y_{j,j} = y_{\sigma_j(2),j} = \ldots = y_{\sigma_j(i),j} = 1$ and $y_{\sigma(i+1),j} = \ldots = y_{\sigma(n),j} = 0$.

The problem can then be modeled as the following MIP.

$$
(\mathrm{P}) = \begin{cases}
\min \sum_{j\in V} \sum_{i\in V\setminus\{j\}} \Delta_{i,j}\, y_{i,j} & & & \\
\text{s.t.} \sum_{j\in V} x_{j,j} \leq K, & & & (8)\\
\sum_{j\in V} x_{i,j} \geq 1 & \forall i \in V, & & (9)\\
\sum_{i\in V} w_i^\ell x_{i,j} \leq W^\ell & \ell = 1,2, & \forall j \in V, & (10)\\
x_{i,j} \leq x_{\pi_j(i),j} & \forall i \in V, i\neq j, & \forall j \in V, & (11)\\
y_{\sigma_j(k),j} \leq y_{\sigma_j(k-1),j} & \forall k \in \{2,\dots,n\}, & \forall j \in V, & (12)\\
x_{i,j} \leq y_{i,j} & \forall i \in V, & \forall j \in V, & (13)\\
x_{i,j} \in \{0,1\} & \forall i \in V, & \forall j \in V, & (14)\\
y_{i,j} \in \{0,1\} & \forall i \in V, & \forall j \in V. & (15)
\end{cases}
$$

The model described hereinabove differs from (1)–(7) by its objective function, which is computed from decision variables $\boldsymbol{y}$, and constraints (12) and (13). Constraints (12) ensure that variables $\boldsymbol{y}$ are consistent, and constraints (13) state that if i is in the district of center j, then the radius of district j is greater than or equal to $d_{i,j}$.

2.3. Extended formulation

In this section, we show how to derive an extended formulation of (P) using Dantzig-Wolfe decomposition.

Given $j \in V$, we define $\mathcal{G}^j$ the set of extreme points of the convex hull of the set of integer solutions to system (10)–(15) associated with j. Note that a solution associated with center j can refer to an empty district, therefore not covering center j. We can rewrite compact model (P) by substituting vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ with an integer combination of the elements of $\mathcal{G}^j$. Given $g \in \mathcal{G}^j$, let $x_{i,j}^g$ (resp. $y_{i,j}^g$) be the value of variable $x_{i,j}$ (resp. $y_{i,j}$) in g. Let $\boldsymbol{\lambda}_j \in \{0,1\}^{|\mathcal{G}^j|}$ be a vector of binary variables such that λ_j^g

is equal to the number of times extreme point g is used in the solution. The new model can be defined as follows.

$$
(\mathrm{P}^M) = \begin{cases}
\min \sum_{j\in V} \sum_{i\in V} \Delta_{i,j} \left(\sum_{g\in\mathcal{G}^j} y^g_{i,j}\, \lambda^g_j \right) & \\
\text{s.t.} \sum_{\substack{j\in V\\ g\in\mathcal{G}^j}} x^g_{j,j}\, \lambda^g_j \leq K, & (\kappa) \quad (16)\\
\sum_{\substack{j\in V\\ g\in\mathcal{G}^j}} x^g_{i,j}\, \lambda^g_j \geq 1 \qquad \forall i \in V, & (\phi) \quad (17)\\
\sum_{g\in\mathcal{G}^j} \lambda^g_j = 1 \qquad \forall j \in V, & (\alpha) \quad (18)\\
\lambda^g_j \in \{0,1\} \qquad \forall g \in \mathcal{G}^j, \forall j \in V. & (19)
\end{cases}
$$

This problem is called the master problem. It has a polynomial number of constraints and an exponential number of variables. A classical way to solve this problem is to use a branch-and-price method. It relies on solving iteratively the continuous relaxation of (P^M). For this purpose, we first solve a restriction of the model to a subset of variables (called the restricted master problem). Then, we need to solve a subproblem to find a non-generated variable corresponding to a negative reduced cost. That subproblem can be decomposed into n subproblems (one for each center candidate). In practice, each subproblem consists in computing the district that corresponds to the smallest reduced cost, with regards to its center candidate. If none of the generated districts correspond to a negative reduced cost, then the current solution for the continuous relaxation of (P^M) is optimal. Otherwise, we add all n generated columns to the restricted master problem (even if the corresponding reduced cost is non-negative).

Let us associate dual variables $\kappa \in \mathbb{R}_-$, $\phi_i \in \mathbb{R}_+$ for all $i \in V$ and $\alpha_j \in \mathbb{R}$ for all $j \in V$ respectively to constraints (16), (17) and (18). We then get, for each center candidate, the following pricing problem.

$$
(\mathrm{P}^j) = \begin{cases}
\min \sum_{i\in V} \Delta_{i,j}\, y_{i,j} - \kappa\, x_{j,j} - \sum_{i\in V} \phi_i\, x_{i,j} - \alpha_j & \\
\Leftrightarrow -\alpha_j - \max \kappa\, x_{j,j} + \sum_{i\in V} \phi_i\, x_{i,j} - \sum_{i\in V} \Delta_{i,j}\, y_{i,j} & \\
\text{s.t.} \sum_{i\in V} w^\ell_i\, x_{i,j} \leq W^\ell \qquad \ell = 1,2, & (20)\\
x_{i,j} \leq x_{\pi_j(i),j} \qquad \forall i \in V, i \neq j, & (21)\\
y_{\sigma_j(k),j} \leq y_{\sigma_j(k-1),j} \qquad \forall k \in \{2,\cdots,|V|\}, & (22)\\
x_{i,j} \leq y_{i,j} \qquad \forall i \in V, & (23)\\
x_{i,j} \in \{0,1\} \qquad \forall i \in V, & (24)\\
y_{i,j} \in \{0,1\} \qquad \forall i \in V. & (25)
\end{cases}
$$

Considering ϕ_i as the profit gained from covering a vertex with some district, this problem amounts to searching the valid district that minimizes its radius while maximizing the profit induced by the covered vertices.

In order to get an integer solution of (P^M), we apply a branch-and-price method. The details of this method are presented in a subsequent section. In the following section, we focus on solving the continuous relaxation of model (P^M). The restricted master problem is solved using a linear programming solver. The main difficulty is to solve the column generation subproblems ; therefore, we propose several algorithms that can solve them efficiently.

3. Methods for solving the column generation subproblem

In this section, we focus on the pricing problem introduced in the previous section. This problem can be decomposed into n pricing subproblems, each of them using a distinct center vertex j in V. Thus, for each vertex j in V, we want to generate an extreme point g in $\mathcal{G}^j$ such that the reduced cost of λ^g_j, the associated decision variable in (P^M), is minimized. To simplify notations, we will drop index j henceforth, since we are always considering a subproblem centered at some vertex j. We will also assume, w.l.o.g., that the subproblem is a maximization problem. Consequently, the profit associated with each node is non-negative in the pricing subproblem. However, when applying the Lagrangian relaxation of the capacity constraints, negative profits may also arise. The problem can be formally stated as the following.

Problem 2 (valid district maximizing the sum of profits minus the radius)**.** *Given a tree $T = (V, A)$ rooted in j, two vectors $\boldsymbol{w^1}$ and $\boldsymbol{w^2}$ in $\mathbb{N}^n$, two vectors $\boldsymbol{d}$, and $\boldsymbol{p}$ in $\mathbb{R}^n_+$, and two integers $W^1 \geq 1$ and $W^2 \geq 1$, find a valid district (U, j) maximizing $\sum_{i\in U} p_i - r(U, j)$.*

This problem is a generalization of two variations of the knapsack problem that have been studied in the literature. The first is the so-called Tree Knapsack Problem, defined by a single knapsack constraint and no radiuses (or, equivalently, radiuses equal to 0). In [10], a branch-and-bound algorithm is proposed to solve this problem while a dynamic programming approach is discussed in [11]. Both methods make use of a depth-first search (DFS) ordering of the items. The second variation is the so-called Penalized Knapsack Problem, which considers a single knapsack constraint and no precedence constraints. In this context, the radiuses are called penalties. In [13], this problem is solved using a dynamic program wherein the items are arranged in order of increasing penalty. However, the precedence constraints change the structure of our problem, preventing us from using a similar ordering.

We designed two solvers for the subproblem wherein the center vertex is given. The first is based on enumerating the relevant $r(U, j)$ values. We then iteratively solve a variation of the knapsack problem with precedence constraints. The second solver directly computes both the set of covered vertices and the district radius simultaneously. Note that, since the Knapsack Problem is weakly NP-hard and the complexity of the dynamic program described in section 3.2 is in pseudo-polynomial time, Problem 2 is weakly NP-hard.

3.1. Subproblem solving by radius enumeration and branch-and-bound

One way to solve the subproblems is to enumerate all potential radiuses in order to obtain a variation of a classical problem from the literature called *Tree Knapsack Problem* (TKP).

Problem 3 (2D knapsack problem with tree-precedence constraints (TKP-2D))**.** *Given a tree $T = (V, A)$ rooted in j, a vector $\boldsymbol{p}$ in $\mathbb{R}^n_+$, two vectors $\boldsymbol{w^1}$ and $\boldsymbol{w^2}$ in $\mathbb{N}^n$, and two integers $W^1 \geq 1$ and $W^2 \geq 1$, find a valid district (U, j) that maximizes $\sum_{i \in U} p_i$.*

There are, at most, n relevant radius values for each center. For each such value r, we can preprocess the instance data to remove each vertex whose distance from the center is greater than r. Given center j and radius r, the subproblem amounts to solving TKP-2D$(T^j; r) - r + \alpha_j$, where TKP-2D$(T^j; r)$ is defined as follows.

$$
(\text{TKP-2D}(T^j; r)) = \begin{cases} \max \sum_{i \in V} p_i\, x_i & \\ \text{s.t.} \sum_{i \in V} w_i^\ell\, x_i \leq W^\ell & \ell = 1, 2, \quad (26) \\ x_i \leq x_{\pi_j(i)} & \forall i \in V, i \neq j, \quad (27) \\ x_i = 0 & \forall i \in V, d_i > r \quad (28) \\ x_i \in \{0, 1\} & \forall i \in V \quad (29) \end{cases}
$$

The problem described hereinabove has already been studied in one dimension in [11, 10]. A dynamic programming algorithm [11] and a branch-and-bound algorithm [10] have been proposed to solve it. These algorithms have been extended to two knapsack constraints in [14].

The branch-and-bound method that we propose is an extension of the algorithm from [10], initially addressing the 1D knapsack problem with tree-precedence constraints.

In [10], the branching scheme of the branch-and-bound algorithm consists in selecting a vertex i in V and constructing two sub-problems: a first subproblem in which node i does not belong to the solution, and a second subproblem in which node i does belong to the solution. The dual bound for each node of the branching tree is obtained by applying a Lagrangian relaxation of the knapsack constraint and performing a linear search to compute the best Lagrange multiplier. Let us first formalize the problem obtained after relaxing the capacity constraint.

Problem 4 (Maximum weight subtree)**.** *Given a tree $T = (V, A)$ rooted in j, and a vector $\boldsymbol{p}$ in $\mathbb{R}^n$, find a valid subtree $T[U]$ that maximizes $\sum_{i \in U} p_i$.*

If $\bar{\mu} \in \mathbb{R}_+$ is the Lagrange multiplier associated with the knapsack constraint, then the Lagrangian relaxation corresponds to the maximum weight subtree problem with $\bar{p}_i = p_i - \bar{\mu} w_i$ where w_i is the weight of item i in the knapsack constraint.

This problem can be solved in linear time [10] using a dynamic program. A state of this program is defined by some vertex $i \in V$ and corresponds to a maximum subtree of $T(i)$. The dynamic program (DP) can then be written as

$$\beta(i) = \max\{0, \bar{p}_i + \sum_{k:\pi(k)=i} \beta(k)\},$$

and the optimal solution is constructed from $\beta(j)$. Using an inverse depth-first search (DFS) ordering to walk the vertices of T, all states of the dynamic program are computed iteratively.

To solve the Lagrangian problem (that is, to find the best multiplier $\bar{\mu}$), one performs a linear search on the value of $\bar{\mu}$. Note that the objective function of the Lagrangian problem is convex and piecewise linear. The linear search starts with a null Lagrange multiplier, and the maximum weight subtree associated with this multiplier. Increasing the value of $\bar{\mu}$ will linearly decrease the value of the Lagrangian function (assuming the current solution is not already feasible) while keeping the same solution until, for some vertex i^* of the solution, profit $\beta(i^*)$ becomes zero (a break point of the piecewise linear function is reached). Vertex i^* is called a *critical item*. The new solution is simply obtained by removing subtree $T(i^*)$ from the previous solution. The procedure is repeated until a solution satisfying the knapsack constraint is obtained. The value of $\bar{\mu}$ used to obtain the last breaking point is the optimal solution of the Lagrangian problem.

The sequence of critical items obtained along the solution of the Lagrangian problem is used to define the successive branching decisions in the branch-and-bound tree.

In the two dimensional case, we consider the same branching scheme and evaluation method. However, as the Lagrangian relaxation is obtained by penalizing two knapsack constraints, the linear search is no longer relevant to find the best Lagrange multipliers. Consequently, we use a descent algorithm to optimize them heuristically. In order to fully take advantage of the branching strategy used in [10], based on a sequence of critical items, the descent algorithm should iteratively remove subtrees rooted in critical items, similarly to the linear search in the one dimensional case.

Let us first formalize the problem obtained after relaxing both capacity constraints. Given multipliers $\boldsymbol{\mu} \in \mathbb{R}^2$, the problem is again equivalent to the maximum weight subtree problem with $\tilde{p}_i = p_i - \boldsymbol{\mu} \cdot \boldsymbol{w}_i$ for all $i \in V$ and can be solved with the same dynamic program.

Now, by restricting the descent algorithm to directions that only increase the Lagrange multipliers, we have that each successive solution is obtained by removing subtrees from the previous one. This property follows from the fact that the values $\tilde{p}_i$ and $\beta(i)$ decrease proportionally to $\boldsymbol{\mu}$. The optimal solution remains identical until at least one of the DP state is such that $\beta(i^*) = 0$ for some vertex i^* of the solution (which defines the length of the step made in the chosen direction), in which case we remove the subtree rooted in the critical vertex i^*.

The sequence of critical vertices induced by the descent algorithm is used as the successive branching decisions in our branch-and-bound algorithm.

Finally, let us describe the descent direction that we use. The most straightforward direction corresponds to the violation of the capacity constraints by the current solution. However, preliminary tests showed that moving along only one capacity constraint at a time, namely the most violated one, leads to better performances.

When (TKP-2D($T^j; r$)) is solved for a given value r, we obtain a primal bound for (P^j), which can be used to prune branch-and-bound nodes for subsequent radius values (sometimes even at the root node). We also obtain a dual bound for any radius $r' < r$ by keeping the same set of vertices and updating the objective function by replacing penalty r with r'. The value obtained is a valid upper bound since it is the value of an optimal solution of a relaxation of the problem associated with radius r' (some vertices of larger radius can still be used, which relaxes the problem).

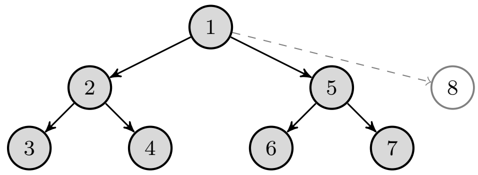
Considering this, we first focus on the primal bound by evaluating the *"expected"* best radius first, *i.e.* the radius that is the most often associated with an optimal solution of subproblem (P^j), before evaluating the other radiuses in decreasing order to produce useful dual bounds.

3.2. Direct subproblem solving with dynamic programming

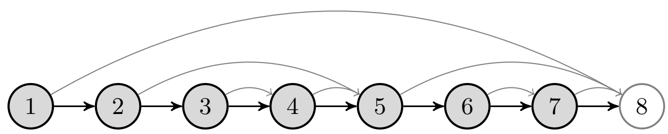
While our branch-and-bound algorithm is based on an enumeration of the relevant radiuses, the dynamic program (DP) presented hereafter solves a subproblem by computing both the set of covered vertices and the district radius simultaneously. First, we present a DP designed for the case where the radius is given, which is an extension of the one presented in [11] in two dimensions. Then, we propose a method to efficiently take the radius into account in the DP's recurrence, as well as several techniques that improve the algorithm's performances.

3.2.1. Dynamic program for the TKP-2D

This dynamic program was originally proposed in [11] in one dimension as a variant of [15], then extended to two dimensions in [14]. We present the dynamic program using our own formalism. In the following, we consider a tree $T = (V, A)$ rooted in j. W.l.o.g., we assume that the vertices of V are indexed according to some depth-first search (DFS) ordering. Under this assumption, if item k is selected, the next choice is related to item $k+1$ (the next in the DFS ordering). If item k is not selected, no vertices in $T(k)$ can be selected. In this case, the next vertex to be considered is $\eta(k) = \min\{i : i > k, i \notin T(k)\}$. If k is the last vertex in the DFS ordering, or if all indices $k' > k$ correspond to descendants of k, then we assume, by convention, that $\eta(k) = n+1$. Figures 1a and 1b represent the relation between the precedence tree and the successorship structure in the dynamic program.



(a) A precedence graph (where node 8 is an artificial node bearing no profit nor weight)



(b) Successorship structure corresponding to the precedence graph in 1a. Arcs $(k, k+1)$ are drawn in black while arcs $(k, \eta(k))$ are grey.

Figure 1: Relation between the precedence graph and the successorship structure.

Building on this precedence structure, we can define the states of the dynamic program by pairs $(k, \boldsymbol{w})$, where $k \in \{1, \dots, n+1\}$ represents the index in the DFS order, and $\boldsymbol{w} \in \{0, \dots, W^1\} \times \{0, \dots, W^2\}$ is a vector representing resource consumptions. Since profits are computed using dual values, they are not integer. Thus, only DP based on weights are viable.

In the following, for any two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ of the same dimensions, $\boldsymbol{u} \leq \boldsymbol{v}$ and $\boldsymbol{u} + \boldsymbol{v}$ are componentwise operations. Let $\boldsymbol{W} = \begin{pmatrix} W^1 \\ W^2 \end{pmatrix}$. Then, the dynamic program can be defined as follows.

$$\psi^{\mathrm{CS}}(k, \boldsymbol{w}) = \begin{cases} 0 & \text{if } k = n+1, \\ \max\{p_k + \psi^{\mathrm{CS}}(k+1, \boldsymbol{w} + \boldsymbol{w}_k), & \text{if } k \leq n \\ \qquad\qquad \psi^{\mathrm{CS}}(\eta(k), \boldsymbol{w})\} & \quad \text{and } \boldsymbol{w} + \boldsymbol{w}_k \leq \boldsymbol{W}, \\ \psi^{\mathrm{CS}}(\eta(k), \boldsymbol{w}) & \text{otherwise,} \end{cases}$$

where $\psi^{\mathrm{CS}}(k, \boldsymbol{w})$ represents the value of the best solution limited to vertices in $\{k, \dots, n+1\}$ and a resource consumption less than or equal to $\boldsymbol{w}$. The optimal solution corresponds to $\psi^{\mathrm{CS}}(1, \begin{pmatrix} 0 \\ 0 \end{pmatrix})$.

Note that the capacity constraints are included in the structure of the graph. Therefore, solving this dynamic program is equivalent to finding a longest path in a directed acyclic graph, where the vertices are states of the program and the arcs represent the options available in the recurrence.

This directed acyclic graph has $\mathcal{O}(nW^1W^2)$ vertices and arcs. A longest path from vertex $(1, \begin{pmatrix} 0 \\ 0 \end{pmatrix})$ can be computed in $\mathcal{O}(|V|W^1W^2)$ time and space. The algorithm of

[11] is a direct application of Bellman's algorithm on this graph, in the case of a single capacity constraint.

On the one hand, using a DFS ordering is motivated by the precedence constraints, which can then be integrated in the recurrence relation. On the other hand, the best DP-based methods for the penalized knapsack problem [16] consider the items in order of increasing penalty, so that the penalty itself does not require adding an explicit dimension to the state space. However, these two orderings are very likely to be contradictory. For example, consider an instance with four items $\{i_0, i_1, i_2, i_3\}$ such that $\pi(i_1) = \pi(i_2) = i_0$, $\pi(i_3) = i_2$, and $r(i_0) < r(i_2) < r(i_1) < r(i_3)$. Then, it is clear that there does not exist a DFS ordering such that the items are arranged in order of increasing radius (penalty). For this reason, the dynamic program in the next section will be based on a DFS ordering of the items.

3.2.2. Extension of the dynamic program to include district radiuses

In this section, we show how to extend the dynamic program defined hereinabove to include district radiuses in the computation of the solution cost. The dynamic program can then be directly used to solve (P^j). For the sake of clarity, let r_i be the radius corresponding to vertex i.

The basic idea of the following DP is to modify ψ^{CS} in such a way that it takes into account the current radius of the partial solution. Now, a state has the form $(k, \boldsymbol{w}, r)$, where k and $\boldsymbol{w}$ are the same as in ψ^{CS}, and r is the radius of the current partial solution. When item k is considered, the radius does not change if k is not selected. If k is selected, the new radius is udpated to value r_k if $r_k > r$. In the latter case, the cost of the state will also be updated to account for the penalty entailed by the new radius.

Given $x \in \mathbb{R}$, let $(x)^+ = \max\{0, x\}$. If $\boldsymbol{x}$ is a vector, then $(\boldsymbol{x})^+$ applies componentwise. We assume, w.l.o.g., that the vertices are indexed according to some DFS ordering.

$$\psi^{\mathrm{R}}(k, \boldsymbol{w}, r) = \begin{cases} 0 & \text{if } k = n+1, \\ \max\Big\{p_k + (r_k - r)^+ + \psi^{\mathrm{R}}(k+1, & \text{if } k \leq n \\ \qquad \boldsymbol{w} + \boldsymbol{w}_k, \max\{r, r_k\}), & \text{ and } \boldsymbol{w} + \boldsymbol{w}_k \leq \boldsymbol{W}, \\ \qquad\qquad \psi^{\mathrm{R}}(\eta(k), \boldsymbol{w}, r)\Big\} & \\ \psi^{\mathrm{R}}(\eta(k), \boldsymbol{w}, r) & \text{otherwise.} \end{cases}$$

In this dynamic program, $\psi^{\mathrm{R}}(1, \begin{pmatrix}0\\0\end{pmatrix}, 0)$ corresponds to an optimal solution for problem (P^j). Since the considered graph has no oriented cycles, we use a label-setting algorithm to solve this problem, *i.e.* we consider the vertices of the graph in a topologic ordering, and keep a set of non-dominated partial solutions for each node. Non-dominated solutions are represented by *labels*. A label ℓ is defined by tuple $(p(\ell), k(\ell), \boldsymbol{w}(\ell), r(\ell))$ containing the cost of the partial path from vertex $(1, \begin{pmatrix}0\\0\end{pmatrix})$, the current position in the DFS ordering, the resource consumptions and the current district radius. The theoretical complexity of this new DP is $\mathcal{O}(|V|^2 W^1 W^2)$ in time and space.

We now need to define a dominance relation that takes the radius into account. Let ℓ_i and ℓ_j be two labels such that $k(\ell_i) = k(\ell_j)$. Label ℓ_i dominates label ℓ_j if and only if at least one of the following dominance relations is met. Clearly, ℓ_i dominates ℓ_j if ℓ_i bears a higher profit than ℓ_j while consuming less resource and being closer to the district center.

Dominance relation 1. *Label ℓ_i dominates label ℓ_j if $p(\ell_i) \geq p(\ell_j)$, $r(\ell_i) \leq r(\ell_j)$, and $\boldsymbol{w}(\ell_i) \leq \boldsymbol{w}(\ell_j)$*

The second dominance relation implies that ℓ_i dominates ℓ_j if ℓ_i bears a better profit and *balances its radius* while consuming less resource than ℓ_j.

Dominance relation 2. *Label ℓ_i dominates label ℓ_j if $p(\ell_i) \geq p(\ell_j)$, $p(\ell_i) - r(\ell_i) \geq p(\ell_j) - r(\ell_j)$, and $\boldsymbol{w}(\ell_i) \leq \boldsymbol{w}(\ell_j)$.*

Proposition 3.1. *Dominance relation 2 is correct.*

Proof. If dominance relation 2 is verified and $r(\ell_i) \leq r(\ell_j)$, then dominance relation 1 is verified as well, and ℓ_i dominates ℓ_j.

Let us then assume that $r(\ell_i) > r(\ell_j)$. ℓ_i is a better label than ℓ_j, since its net profit $p(\ell_i) - r(\ell_i)$ is higher than that of ℓ_j, while its resource consumption is lower. In order to justify maintaining ℓ_j, we have to assume that it will be combined with a set of vertices I. Let $\oplus$ be the operation that adds a set of vertices to some label. Since $k(\ell_i) = k(\ell_j)$ and $\boldsymbol{w}(\ell_i) \leq \boldsymbol{w}(\ell_j)$, we can also combine I and ℓ_i, then compare the resulting new labels $(\ell_i \oplus I)$ and $(\ell_j \oplus I)$. Since the net profit of these two new labels is $\Big(p(\ell_i) + p(I) - \max\{r(\ell_i), r(I)\}\Big) \geq \Big(p(\ell_j) + p(I) - \max\{r(\ell_j), r(I)\}\Big)$, then $(\ell_i \oplus I)$ is still better than $(\ell_j \oplus I)$. This proves that ℓ_i dominates ℓ_j. □

Dominance relation 2 is at least as strong as 1 and does not involve heavier computations. For this reason, it is natively included in our algorithm.

The validity of the dynamic program builds on the fact that it considers the vertices iteratively according to a DFS ordering based on the precedence graph. Any such ordering guarantees the consistency of indices $k+1$ and $\eta(k)$, for any vertex $k \in V$. Note that there exists an exponential number of such DFS orderings. Among these orderings candidates, we break ties by selecting the one in which vertices are also arranged in order of decreasing radius induced by their subtree.

For specific instances, using the concept of subtree radiuses in a DFS ordering leads to better performances in practice for the DP. An explanation for this behavior is that the DP states are associated with their final radius earlier in the recurrence relation, thus generating labels with higher costs that can be eliminated earlier.

3.2.3. Lagrangian filtering method

The dynamic program described in the previous section induces a pseudo-polynomial number of states. The number of labels that are required to solve the program can be huge in some cases, since the number of dimensions is a limiting factor in the algorithm's ability to remove dominated labels. The objective of our filtering method is to detect state transitions that can never be associated with an optimal solution. For this purpose, we use a technique, derived from the SSDP method (see [17], for example), that solves a relaxation, filters out some transitions, and then reintroduces the constraints that were relaxed. In our implementation of this method, we use a Lagrangian relaxation of the capacity constraints.

Given $\boldsymbol{\zeta} \in \mathbb{R}^2$ a Lagrange multiplier vector, we compute the Lagrangian profit $\hat{p}_k$ of each item k, that is, $\hat{p}_k = p_k - \boldsymbol{\zeta} \cdot \boldsymbol{w}_k$. The recurrence can then be defined as follows.

$$\psi^{\mathrm{LR}}(k,r) = \max \Big\{ \hat{p}_k + (r_k - r)^+ + \psi^{\mathrm{LR}}(k+1, \max\{r, r_k\}), \psi^{\mathrm{LR}}(\eta(k), r) \Big\} \quad (30)$$

Let us now consider the graph corresponding to DP (30). Each node of this graph is a state of the DP. The arcs correspond to the potential decisions from each state (selecting an item or not). Figure 2 represents an example of such a graph, where the arc costs are defined as follows. Each vertex (k,r) has two outgoing arcs. Arc $a = \big((k,r), (k+1, \max\{r, r_k\})\big)$, represents selecting item k in the solution, and its cost $\tilde{p}_a$ is $\hat{p}_k + (r_k - r)^+$. Arc $a' = \big((k,r), (\eta(k), r)\big)$ represents dismissing this item, which does not induce any additional cost, *i.e.* $\tilde{p}_{a'} = 0$.

The complexity of this DP is in $\mathcal{O}(|V|^2)$ in time and space, since each of the n vertices is associated with at most one state per relevant radius.

Let PB be some primal bound for the value of an optimal solution. For example, that bound could have been inferred from a feasible solution. Given a vector of Lagrange multipliers, we compute, for each vertex (i,r) in the state graph, $\omega^-((i,r))$ and $\omega^+((i,r))$ defined, respectively, as the value of a longest path from vertex $(1,0)$ to vertex (i,r) and the value of the longest path from vertex (i,r) to vertex $(n+1,0)$. Given arc $a = ((i,r),(j,r'))$, if $\omega^-((i,r)) + \tilde{p}_a + \omega^+((j,r')) < PB$, then a is not associated with an optimal solution. Note that this bound is still valid in the original space with the capacity constraints. In practice, when we filter out the outgoing arc of vertex (k,r) that corresponds to selecting item k, we will not consider solutions constructed from label $(p,k,\boldsymbol{w},r)$ for any values p and $\boldsymbol{w}$. The same applies to outgoing arcs that correspond to disregarding a item.

Lagrange multipliers $\boldsymbol{\zeta}$ are computed using a standard subgradient algorithm. Let $\hat{\boldsymbol{p}}^t \in \mathbb{R}^n$ be the vector of Lagrangian profits, that is, $\hat{p}^t_k = p_k - \boldsymbol{\zeta}^t \cdot \boldsymbol{w}_k$, and let $\boldsymbol{x}^t$ be the optimal solution obtained after iteration t of the subgradient algorithm. The associated subgradient di-

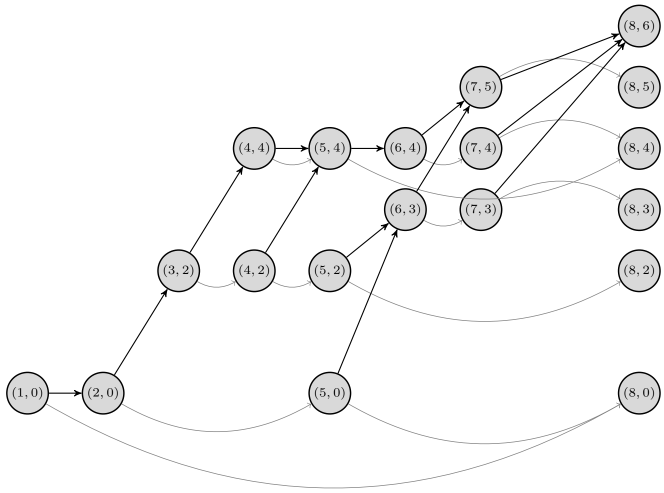


Figure 2: Graph representing the dynamic program induced by Lagrangian relaxation. States are defined by both a position in the DFS ordering and a radius. Bold arcs correspond to selecting the current item. Grey arcs correspond to not selecting the current item.

rection ∇^t is defined by the capacity constraints violation, $\nabla^t = \begin{pmatrix} \boldsymbol{w}^1 \cdot \boldsymbol{x}^t - W^1 \\ \boldsymbol{w}^2 \cdot \boldsymbol{x}^t - W^2 \end{pmatrix}^+$ and the step size s^t follows Polyak's step size, $s^t = 2 \cdot \frac{0.1(\hat{\boldsymbol{p}}^t \cdot \boldsymbol{x}^t)}{\|\nabla^t\|_2^2}$, where $0.1(\hat{\boldsymbol{p}}^t \cdot \boldsymbol{x}^t)$ is an approximation of the optimality gap between the current multipliers $\boldsymbol{\zeta}^t$ and the optimal multipliers $\boldsymbol{\zeta}^*$. The algorithm terminates after two non-improving iterations, *i.e.* iterations where $\hat{\boldsymbol{p}}^t \cdot \boldsymbol{x}^t \leq \hat{\boldsymbol{p}}^t \cdot \boldsymbol{x}^{t-1}$.

3.2.4. Completion bounds: filtering out labels according to their cost

Each label $(p,k,\boldsymbol{w},r)$ of the dynamic program corresponds to a partial solution that is feasible with regards to the capacity constraints and the precedence constraints of the pricing subproblem. In a dynamic program, the main way to remove labels is to check dominance relations. However, it is also possible to remove labels using dual bounds, as we would in a branch-and-bound algorithm. Given some primal bound PB and some label ℓ, filtering out label ℓ is valid if $p(\ell) + DB(\ell) \leq PB$, where $DB(\ell)$ corresponds to some dual bound on the value that can be gained from ℓ to some terminal state, thus completing label ℓ.

The strength of this test depends on the quality of dual bound $DB(\ell)$. In order to save computation time, the method that we use recycles computations from the Lagrangian filtering method that was described previously. Let $\boldsymbol{\zeta} \in \mathbb{R}^2_+$ be a vector of Lagrangian multipliers. Given label ℓ defined by tuple $(p(\ell), k(\ell), \boldsymbol{w}(\ell), r(\ell))$, let ℓ' be its projection, by relaxation of the capacity constraints, defined by triple $(p(\ell) - \boldsymbol{\zeta} \cdot \boldsymbol{w}(\ell), k(\ell), r(\ell))$, and let $v_{\boldsymbol{\zeta}}(\ell')$ be the largest value of a path from the vertex bearing ℓ' to sink $(n+1, 0)$. A dual bound $DB(\ell)$ on the best solution that can be constructed from ℓ is given by $DB(\ell) = v_{\boldsymbol{\zeta}}(\ell') + \boldsymbol{\zeta} \cdot (\boldsymbol{W} - \boldsymbol{w}(\ell))$. Given a pair $(k(\ell), r(\ell))$

and a residual capacity $\boldsymbol{W}_r = \boldsymbol{W} - \boldsymbol{w}(\ell)$, we could compute optimal Lagrange multipliers $\boldsymbol{\zeta}^*_{\boldsymbol{W}_r}$. However, a less time-consuming approach consists in reusing $\bar{\boldsymbol{\zeta}}^*$ the multipliers that were obtained at the end of the subgradient algorithm described in the previous section. This technique will be called *"completion bound"* in the computational experiments.

4. Speeding up the convergence of the branch-and-price algorithm

4.1. Generating initial columns

In essence, our column generation algorithm is based on the primal simplex algorithm. One of the bottlenecks of this algorithm is to find an initial feasible basis. In order to address this issue, one generally uses an artificial basis, made from a set of artificial columns that do not exist in the original linear program but whose cost is prohibitive, so that they cannot be part of an optimal basis with a positive coefficient. This phase might require a significant number of column generation iterations before removing all artificial columns from the basis. Thus, by generating a set of random columns that do not take reduced costs into account — since the latter are biased by the cost of the basic artificial columns —, we are able to reduce the time spent in phase one of the simplex algorithm.

In order to generate such columns, we apply the following greedy heuristic for every center vertex. We start from the empty solution, *i.e.* the solution that does not cover any vertex. If there are no vertex that can be appended to the current solution, stop the method. Otherwise, add a vertex that is 1) the closest to the center, 2) not covered yet, and 3) induces a new solution that still verifies both capacity constraints. This algorithm generates districts that cover vertices close to their centers. It can be adapted to generate sparse districts instead, which adds more variety to the set of initial columns.

4.2. Branch-and-price

In order to obtain an integer solution for the master problem (P^M), we use a branching algorithm. In each branching node, we solve the continuous relaxation of problem (P^M), to which we add new constraints representing the branching decisions.

The branching decisions are based on the variables $x_{i,j}$ from compact model (P). Given vertices i and j, adding constraint $x_{i,j} \geq 1$ (resp. $x_{i,j} \leq 0$) is equivalent to adding constraint $\sum_{g\in\mathcal{G}^j} x^g_{i,j}\lambda^g_j \geq 1$ (resp. $\sum_{g\in\mathcal{G}^j} x^g_{i,j}\lambda^g_j \leq 0$) to problem ($\mathrm{P}^M$). As a consequence, a new dual variable, associated with this branching constraint, is added to the objective function of subproblem (P^j). Note that (P^j) might still generate columns that do not verify the branching constraints depending on the dual values of the constraints it violates. In this case, the column is added to the restricted master problem, and may only enter the basis with a null value. The algorithm branches on the variable $x_{i,j}$ whose value is still very fractional, *i.e.* close to 0.5, after solving the parent node. Finally, the branching nodes are explored according to a *Best-first* search ordering. Another way of branching would be to apply the branching decisions to the subproblem directly. This could be done by cleaning up the master problem and modifying the trees in the subproblems. We chose to branch in the master problem instead, since this method entails no modification in the subproblem (it just changes the cost of the arcs). As such, we do not have to address inconsistency issues such as branching decisions implying that for some $i, j, k \in V$ with i predecessor of k in T^j, $x_{i,j} = 0$ while $x_{k,j} = 1$, which would make subproblem (P^j) infeasible.

Column generation is known to suffer from convergence issues, known as the *tailing-off effect*. In order to address this weakness, we use the stabilisation technique from [18] that substitutes the dual solution for (P^M) by a smoothed dual solution. Let $(\kappa, \boldsymbol{\phi}, \boldsymbol{\alpha})^t$ be the dual solution at iteration t, and let $(\hat{\kappa}, \hat{\boldsymbol{\phi}}, \hat{\boldsymbol{\alpha}})$ be the dual solution associated with the best lower bound known for (P^M) (see below for the computation of this bound). Then, the smoothed solution that we use in the pricing subproblems is given by the following convex combination: $(\tilde{\kappa}, \tilde{\boldsymbol{\phi}}, \tilde{\boldsymbol{\alpha}})^t = \gamma(\hat{\kappa}, \hat{\boldsymbol{\phi}}, \hat{\boldsymbol{\alpha}}) + (1-\gamma)(\kappa, \boldsymbol{\phi}, \boldsymbol{\alpha})^t$. We use the implementation described in [19], which automatically tunes parameter γ during the column generation process.

The so-called *Lagrangian lower bound* is based on the Lagrangian relaxation of constraints (8) and (9) in (P) where dual values κ and $\boldsymbol{\phi}$ are respectively used as Lagrangian mutipliers for (8) and (9). When this relaxation is applied, each center is now related to an independent supbroblem. Therefore, for any values κ and $\boldsymbol{\phi}$ of proper sign, $L(\kappa, \boldsymbol{\phi}) = K\kappa + \sum_{i\in V}\phi_i + \sum_{j\in V}\min_{g\in\mathcal{G}^j}\{\sum_{j\in V}\sum_{i\in V}\Delta_{i,j}y^g_{i,j} - \kappa\sum_{j\in V}x_{j,j} - \sum_{i\in V}\phi_i\sum_{j\in V}x^g_{i,j}\}$ is a valid lower bound for an optimal solution of (P). Now, let $OPT(RMP)$ be the optimum of the current reduced master problem and $RC(SP_j)$ be the best reduced cost of a variable generated by subproblem j at the current iteration. By strong duality, $OPT(RMP) = K\kappa^* + \sum_{i\in V}\phi^*_i + \sum_{j\in V}\alpha^*_j$, where κ^*, ϕ^* and α^* are optimal dual values for RMP. The best reduced cost related to center j is $RC(SP_j) = \min_{g\in\mathcal{G}^j}\{\sum_{j\in V}\sum_{i\in V}\Delta_{i,j}y^g_{i,j} - \kappa^*\sum_{j\in V}x^g_{j,j} - \sum_{i\in V}\phi^*_i\sum_{j\in V}x^g_{i,j} - \alpha^*_j\}$. Computing $OPT(RMP) + \sum_{j\in V}RC(SP_j)$, one obtains $L(\kappa^*, \boldsymbol{\phi}^*) + \sum_{j\in V}\alpha^*_j - \sum_{j\in V}\alpha^*_j = L(\kappa^*, \boldsymbol{\phi}^*)$. Therefore, $OPT(RMP) + \sum_{j\in V}RC(SP_j)$ is a valid lower bound for the problem.

5. Computational experiments

The purpose of our experiments is twofold: to empirically determine which oracle works best in the context of column generation, and to validate the hypothesis that the extended formulation leads to better performances than the original compact formulation.

5.1. *Instances solved and hardware configuration*

Two sets of instances are used to measure the performances of our methods. Instance set `exeo` is composed of 13 instances built from real-life data provided by a French company, which develops and publishes software solutions for route optimization and geolocation of household waste trucks. These instances range from 31 to 245 collection points. Instance set `cvrp` was randomly generated using 16 CVRP (Capacitated Vehicle Routing Problem) instances from the TSPLIB. Each instance from the TSPLIB was used to generate 25 new instances for our problem, resulting in a benchmark of 400 instances ranging from 7 to 262 collection points.

In order to generate several instances from one TSPLIB instance, we added some random noise to the arc lengths as follows. Given (i, j) an arc of length $c_{i,j}$, we define the length $c'_{i,j} = c_{i,j} + r$, with r a random real number generated according to a uniform distribution in $[0, (c_{i,j})^{1.5}]$. In addition, we generated the second capacity constraint by keeping the same capacity (right-hand side) and shuffling the resource consumptions according to a random permutation.

Overall, the item resource consumptions can go as high as 4 100 and the district capacities range from 3 to 14 400, the average capacity being around 1 800.

The experiments were performed on PlaFRIM (Plateforme Fédérative pour la Recherche en Informatique et Mathématiques), using *Dodeca-core Haswell Intel® Xeon® E5-2680 v3 @ 2,5 GHz* processors and 128Go of RAM.

The linear programs and compact mixed-integer programs were solved using IBM ILOG CPLEX Optimization Studio 12.6.0 [20]. We used the branch-and-price implementation from the BaPCod 2.0.0 framework [21], which manages the branch-and-price search tree and provides a built-in stabilization.

5.2. *Analysis of the branch-and-bound oracle*

Our first numerical experiments focus on the branch-and-bound method that we use to solve the column generation subproblem. Our objective is to find the best parameters for this algorithm. Specifically, there are two components that can impact the performances of the algorithm in a significant way: 1) the method used to solve the Lagrangian relaxation and 2) the order in which the radiuses are considered.

These results are described in Table 1. First, we compare two descent directions. Direction `PROP` corresponds to the capacity constraint violation projected onto $\mathbb{R}^2_+$, while direction `STEEP` only increases the penalty associated with the most violated capacity constraint. Then, we combine the best of these two directions with a special ordering for the radiuses. In this ordering, we first solve the radius that is most frequently associated with an optimal solution for the pricing subproblem before proceeding with the remaining radiuses in order of decreasing distance. In the following, this last configuration is referred to as `STEEP+BR`. For each configuration and for each instance set, we report **time** the average time in seconds spent solving an instance to optimality (limited to two hours), **nodes/cg** the average number of branching nodes that were generated by the oracle during that time and **solved** the number of instances that were solved in less that two hours.

It transpires from these experiments that the direction we apply has a significant impact on the algorithm performances, since direction `STEEP` requires 43% less branching nodes in average compared to direction `PROP`. The improved radius ordering allows us to further reduce the number of branching nodes by another 80%. This reduction comes from the fact that this particular ordering allows us to quickly find a good primal bound for the subproblem that is then used to filter out other radiuses. In the next sections, the results for the branch-and-bound oracle correspond to configuration `STEEP+BR`.

5.3. *Analysis of the dynamic programming oracle*

In this section, we compare several parameters for the dynamic program in the context of column generation. The most important is the choice of the methods to eliminate states of the dynamic program, namely dominance relations, Lagrangian filtering, and completion bounds.

The default strategy is referred to as `base`. In this configuration, we use dominance relation 1 and we use neither Lagrangian filtering nor completions bounds. Furthermore, the items are ordered according to an arbitrary DFS ordering. Building on this configuration, we can enrich the algorithm by using the following options: `dom` corresponds to applying dominance relation 2 ; `filt` corresponds to the Lagrangian filtering method described in section 3.2.3 ; `sort` refers to a DFS ordering of the items that takes into account the radiuses of their subtrees ; options `sgCB` and `stCB` are associated with Lagrangian bounds computed from the Lagrangian relaxation of the capacity constraints. The difference between these last two is that `sgCB` uses the Lagrangian multipliers obtained at the end of the Lagrangian filtering method, while `stCB` sets one multiplier to 0 and computes the optimal multiplier for the other dimension. In strategy `all`, we enable every option.

We also compare the impact of removing one of these features from `all`: `w/o dom` use options `filt`, `sort`, `sgCB` and `stCB` but does not use `dom`.

The performances associated with each configuration are recorded in Table 2. For each instance set, we report **time** the average time in seconds spent solving an instance to optimality (limited to two hours), **labels/cg** the average number of labels of the dynamic program that were generated during that time and **solved** the number of instances that were solved in less that two hours.

The experiments confirm that all proposed techniques significantly improve the performances of the dynamic programming algorithm. In particular, two configurations are

Instance	PROP			STEEP			STEEP+BR		
	time	**nodes/cg**	**solved**	**time**	**nodes/cg**	**solved**	**time**	**nodes/cg**	**solved**
exeo/031	≈ 0	776	1/1	≈ 0	344	1/1	≈ 0	284	1/1
exeo/032	≈ 0	46	1/1	≈ 0	46	1/1	≈ 0	19	1/1
exeo/054	1	15	1/1	≈ 0	15	1/1	≈ 0	10	1/1
exeo/075	67	51	1/1	70	51	1/1	65	31	1/1
exeo/087	31	279	1/1	31	279	1/1	26	185	1/1
exeo/102	50	109	1/1	50	109	1/1	48	71	1/1
exeo/109	1 471	4 328	1/1	1 478	4 328	1/1	765	1 648	1/1
exeo/125	7 200	1 860	0/1	7 200	1 859	0/1	7 200	720	0/1
exeo/162	212	1 697	1/1	212	1 697	1/1	178	1 127	1/1
exeo/163	7 200	888	0/1	7 200	893	0/1	7 200	418	0/1
exeo/171	7 200	7 978	0/1	7 200	7 980	0/1	7 200	1 729	0/1
exeo/174	7 200	8 000 257	0/1	1 288	8 963	**1/1**	635	1 998	**1/1**
exeo/245	7 200	6 494 471	0/1	7 200	39 808	0/1	7 200	28 641	0/1
	time	**nodes/cg**	**solved**	**time**	**nodes/cg**	**solved**	**time**	**nodes/cg**	**solved**
cvrp/att48	3 210	448	**17/25**	2 909	251	**17/25**	3 182	188	16/25
cvrp/eil7	≈ 0	107	25/25	≈ 0	92	25/25	≈ 0	46	25/25
cvrp/eil13	≈ 0	46	25/25	≈ 0	46	25/25	≈ 0	46	25/25
cvrp/eil22	≈ 0	475	25/25	≈ 0	408	25/25	≈ 0	143	25/25
cvrp/eil23	3	952	25/25	3	961	25/25	3	930	25/25
cvrp/eil30	1	625	25/25	1	529	25/25	1	197	25/25
cvrp/eil31	2	1 802	25/25	2	1 626	25/25	2	1 292	25/25
cvrp/eil33	32	3 514	25/25	23	2 556	25/25	12	893	25/25
cvrp/eil51	316	3 084	25/25	345	2 134	25/25	212	916	25/25
cvrp/eilA101	6 810	22 557	1/25	7 018	10 782	1/25	7 200	4 016	1/25
cvrp/eilA76	6 161	7 755	6/25	6 183	5 022	6/25	5 385	1 228	**12/25**
cvrp/eilB101	6 970	15 405	2/25	6 915	11 631	2/25	6 717	3 117	2/25
cvrp/eilB76	6 962	10 986	1/25	6 958	8 736	1/25	5 910	1 755	**9/25**
cvrp/eilC76	5 148	6 438	9/25	4 914	3 407	**11/25**	4 651	1 524	**11/25**
cvrp/eilD76	5 654	7 649	8/25	5 173	3 434	9/25	4 762	1 300	**13/25**
cvrp/gil262	7 355	792 422	0/25	7 200	404 621	0/25	7 200	26 934	0/25
Total (/400)			**244**			**247**			**264**

Table 1: Performances of the branch-and-bound oracle

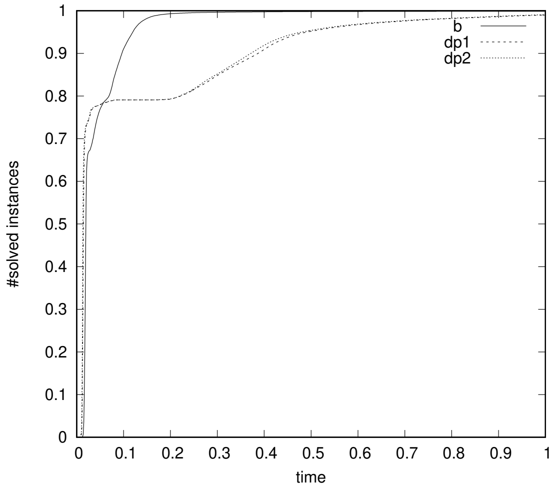


Figure 3: exeo/075 walltimes

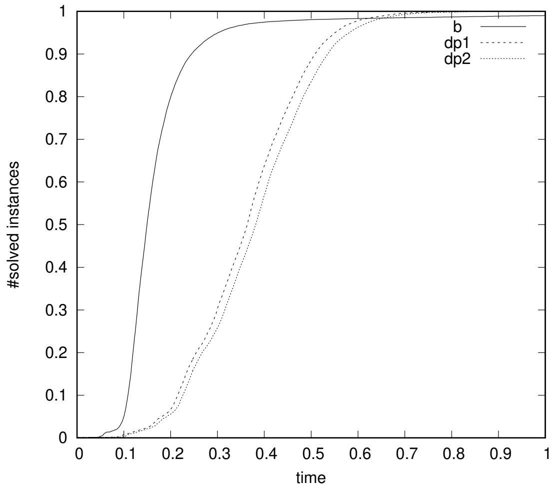


Figure 4: exeo/162 walltimes

standing out: configuration `all` is best suited to solve instances from `exeo` set, while configuration `w/o sort` is more adapted to `cvrp` instances. In both configurations, the algorithm generates about 90% less labels than configuration `base`.

In the remainder, results for the dynamic programming oracle correspond to configurations `all` (referred to as `DP1`) and `w/o sort` (referred to as `DP2`).

5.4. Comparison of the subproblem oracles

We now compare both approaches for solving the column generation subproblem that we described previously: branch-and-bound and dynamic programming. To do so, we use two different experimental setups. In the first set of experiments, we compare the performances of the branch-and-price algorithm when combined with one of the two oracles. However, this comparison is biased, as the subproblems solved by both oracles might be different: when the subproblem supports several equivalent solutions, the oracles might generate different columns, which changes the behavior of the branch-and-price algorithm. As a consequence, we propose a second way to compare the oracles. We first saved the pricing instances solved during the branch-and-price algorithm while using the branch-and-bound oracle. Then, we solved every pricing instance independently using one of the oracles, outside of the branch-and-price scheme. Table 3 and figures 3–14 summarize these computational experiments.

In Table 3, we report **time** the average time in seconds spent solving (P) using the branch-and-price algorithm, **t_MP** the average time in seconds spent solving (P^M) — as opposed to solving the subproblems (P^j) —, and **solved** the number of instances that were solved in less that two hours.

When we focus on the branch-and-price performances, the branch-and-bound oracle is more efficient than the dynamic programming oracles, since it is 1.7 times faster on the `exeo` instances and can solve 21 additional `cvrp`

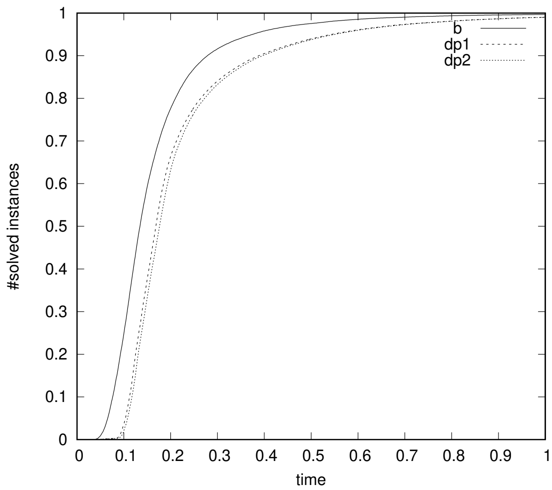


Figure 5: exeo/174 walltimes

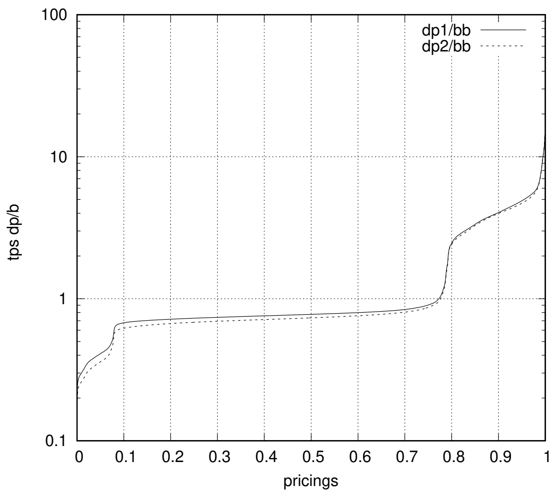


Figure 6: exeo/075 time ratios

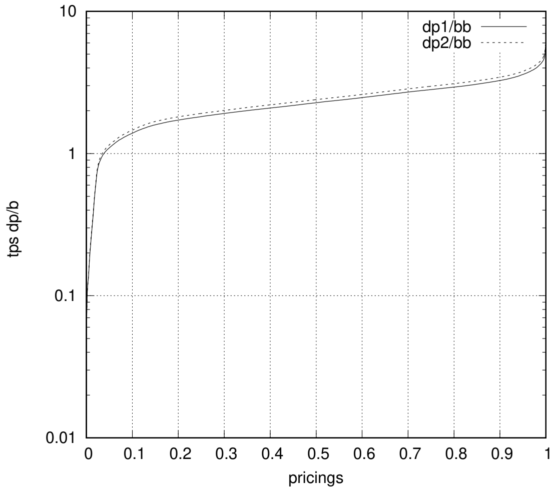


Figure 7: exeo/162 time ratios

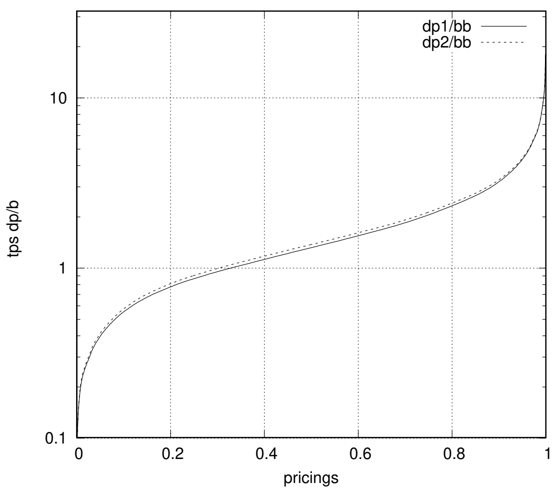


Figure 8: exeo/174 time ratios

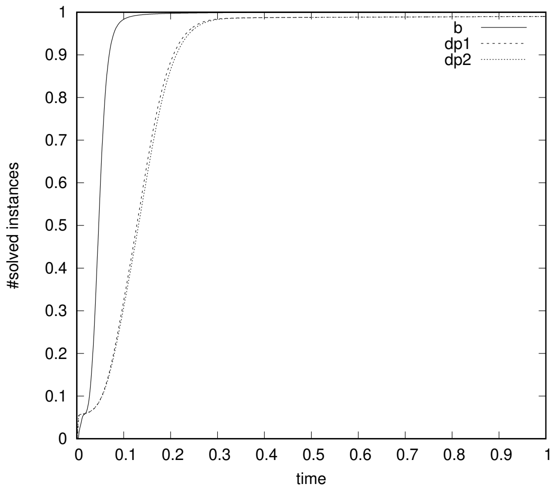


Figure 9: cvrp/att48 walltimes

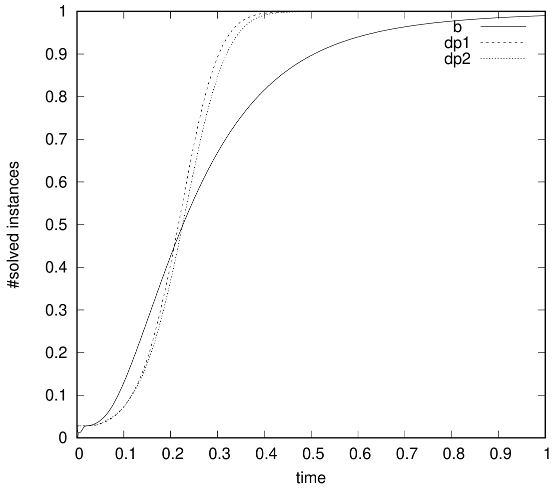


Figure 10: cvrp/eilB76 walltimes

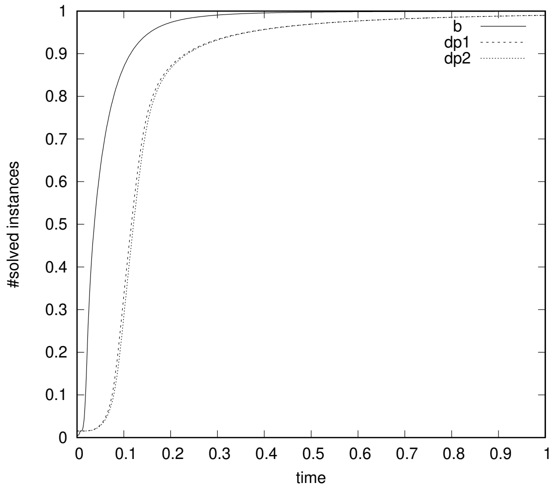


Figure 11: cvrp/eilD76 walltimes

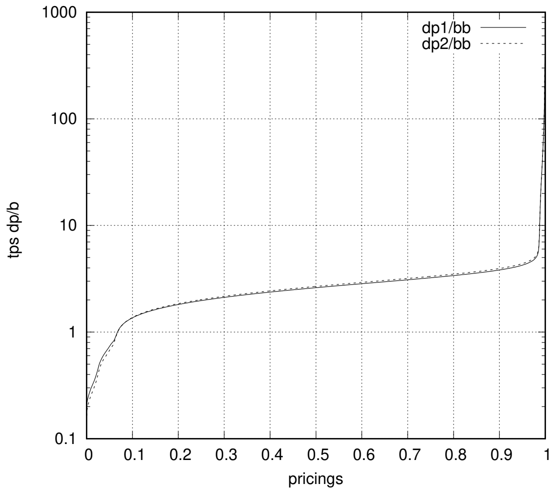


Figure 12: cvrp/att48 time ratios

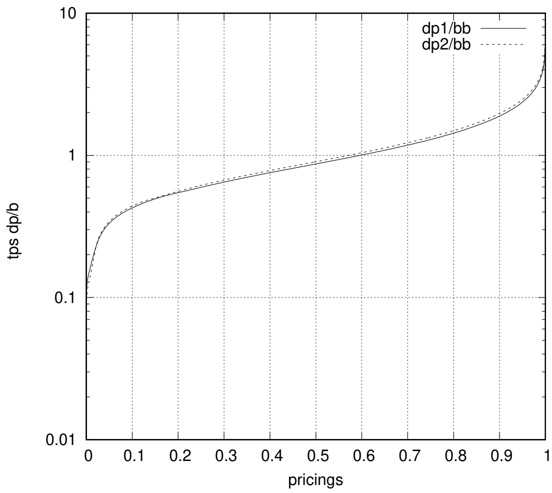


Figure 13: cvrp/eilB76 time ratios

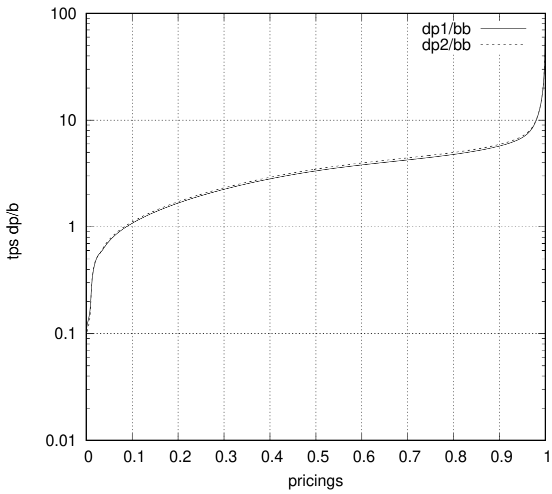


Figure 14: cvrp/eilD76 time ratios

instances. However, both methods exhibit performances that have the same order of magnitude for our test instances.

We now offer a more accurate comparison of the oracles' performances. On the one hand, figures 3–5 and 9–11 represent the number of pricing instances that were solved by each oracle under a given time limit. For example, in Figure 3, the curve corresponding to the branch-and-bound oracle shows that 90% of the pricing instances are solved in less than 0.1 time unit, while the remaining 10% are solved under 0.4 time units. In comparison, 20% of the pricing instances require at least 0.2 time units when we use a dynamic programming based oracle. Thus, the higher the curve, the more efficient the algorithm.

Based on these experiments, we can infer that the branch-and-bound oracle solves generally more instances than any dynamic programming oracle, given some time limit. Note, however, that the dynamic program seems more robust for solving pricing instances related to `cvrp/eilB76` (Figure 10).

On the other hand, figures 6–8 and 12–14 represent the solving time ratios between a dynamic programming oracle and the branch-and-bound oracle. We can then see the proportion of instances for which the dynamic programming oracle is faster than the branch-and-bound method, as well as the associated factor of speed. For example, Figure 6 shows that the dynamic programming oracles are faster than the branch-and-bound oracle more than 75% of the time, and can be as much as 9 times faster than the branch-and-bound oracle. By contrast, the branch-and-bound oracle is at least 5 times faster for solving about 20% of the pricing instances.

As in the previous experiments, the branch-and-bound oracle outperforms the dynamic programming method, except for the pricing instances based on `exeo/075` (Figure 6) and `cvrp/eilB76` (Figure 13). In Appendix A, we reported the diagrams associated with additional experiments using the same framework.

In the following experiments, we will only use the branch-and-bound oracle for solving the pricing subproblem.

5.5. *Comparison of the compact formulations and the extended formulation*

In this set of experiments, we compare the performances of our column generation method to those obtained by the compact formulations, which are solved using a commercial solver (CPLEX).

Table 4 compiles the performances of compact models (P_0) and (P) and the extended formulation. In this table, we report **time_tot** the total time in seconds to solve an instance to optimality (or - if the algorithm did not terminate in less than 2 hours), **time_root** the time spent solving the root of the branching tree, which corresponds to a continuous relaxation of the model. The optimality gap, referred to as **gap_opt**, is $\approx 0\%$ iff the algorithm did not terminate in less than 2 hours. If the algorithm did not even find any feasible solution, then this gap is equal to $+\infty$. Column **gap_root** indicates the quality of the continuous relaxation of each model compared to the best known primal solution. Note that we know the optimal value for each `exeo` instance except `exeo/163` and `exeo/171`. Finally, **nb_bb_nodes** indicates the number of branching nodes that were evaluated, and **time_per_node** is the average time in seconds for each of these nodes. Regarding the `cvrp` instances, we only report the number of instances that were solved to optimality in less than 2 hours.

Clearly, the column generation algorithm can solve the instances more efficiently than the compact models combined with a generic solver. Even if evaluating a branching node is generally more time-consuming in the extended formulation, the quality of the continuous relaxation allows us to dramatically reduce the number of branching nodes that are required to solve the instances.

Note that, for instance `exeo/245`, the optimality gap is smaller at the end when we solve compact model (P) than when we solve the extended formulation. As a consequence, column generation methods seem less fit for this instance as the solving time for the restricted master problem is excessively high.

6. Conclusion

In this document, we studied a new covering problem where vertices of a graph are covered by rooted subtrees. We proposed several mathematical models. The most efficient model relies on column generation, for which we proposed different algorithms. To solve the pricing subproblem, the specialized branch-and-bound algorithm we developed is better on average than our state-of-the-art dynamic programming approach. However, there exists a significant number of instances of the subproblems for which the dynamic programming algorithm is faster.

In order to solve larger instances, valid inequalities could be added to the model in order to strenghten the dual bound in the branch-and-price algorithm. However this could significantly complicate the subproblem oracle, and a thorough study is needed to produce an efficient algorithm.

Several other variants of covering problems using trees are also interesting to consider, for examples problems where one would like to minimize the sum of distances between each pair of vertices of each cluster. A similar decomposition method can be applied, but the subproblem would be much more challenging to tackle.

7. Acknowledgments

We would like to thank the anonymous referees for their valuable comments, which helped us to improve the quality of the manuscript.

Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux, CNRS and ANR in accordance to the programme d'investissements d'Avenir (see https://www.plafrim.fr/).

References

[1] S. W. Hess, J. B. Weaver, H. J. Siegfeldt, J. N. Whelan, P. A. Zitlau, Nonpartisan political redistricting by computer, Operations Research 13 (6) (1965) 998–1006. doi:10.1287/opre.13.6.998.

[2] A. Mehrotra, E. L. Johnson, G. L. Nemhauser, An optimization based heuristic for political districting, Management Science 44 (8) (1998) 1100–1114. doi:10.1287/mnsc.44.8.1100.

[3] A. A. Zoltners, P. Sinha, Sales territory alignment: A review and model, Management Science 29 (11) (1983) 1237–1256. doi:10.1287/mnsc.29.11.1237.

[4] R. Z. Ríos-Mercado, E. Fernández, A reactive grasp for a commercial territory design problem with multiple balancing requirements, Computers & Operations Research 36 (3) (2009) 755–776. doi:10.1016/j.cor.2007.10.024.

[5] M. G. Elizondo-Amaya, R. Z. Ríos-Mercado, J. A. Díaz, A dual bounding scheme for a territory design problem, Computers & Operations Research 44 (2014) 193–205. doi:10.1016/j.cor.2013.11.006.

[6] S. Hanafi, A. Freville, P. Vaca, Municipal solid waste collection: An effective data structure for solving the sectorization problem with local search methods, INFOR: Information Systems and Operational Research 37 (3) (1999) 236–254. doi:10.1080/03155986.1999.11732383.

[7] L. Muyldermans, D. Cattrysse, D. Van Oudheusden, District design for arc-routing applications, Journal of the Operational Research Society 54 (11) (2003) 1209–1221. doi:10.1057/palgrave.jors.2601626.

[8] J. Kalcsics, Districting problems, in: Location Science, 2015, pp. 595–622. doi:10.1007/978-3-319-13111-5_23.

[9] L. Silva de Assis, P. M. França, F. L. Usberti, A redistricting problem applied to meter reading in power distribution networks, Computers & Operations Research 41 (2014) 65–75. doi:10.1016/j.cor.2013.08.002.

[10] G. Cho, D. X. Shaw, The critical-item, upper bounds, and a branch-and-bound algorithm for the tree knapsack problem, Networks 31 (1998) 205–216. doi:10.1002/(SICI)1097-0037(199807)31:4<205::AID-NET1>3.0.CO;2-H.

[11] G. Cho, D. X. Shaw, A depth-first dynamic programming algorithm for the tree knapsack problem, INFORMS Journal on Computing 9 (1997) 431–438. doi:10.1287/ijoc.9.4.431.

[12] S. Elloumi, M. Labbé, Y. Pochet, A new formulation and resolution method for the p-center problem, INFORMS Journal on Computing 16 (1) (2004) 84–94. doi:10.1287/ijoc.1030.0028.

[13] A. Ceselli, G. Righini, An optimization algorithm for a penalized knapsack problem, Operations Research Letters 34 (4) (2006) 394–404. doi:10.1016/j.orl.2005.06.001.

[14] F. Clautiaux, J. Guillot, Y. Magnouche, P. Pesneau, D. Trut, F. Vanderbeck, Clustering problem for waste collection, Tech. rep., University of Bordeaux, INRIA Research team ReAlOpt. (2014).

[15] D. S. Johnson, K. A. Niemi, On knapsacks, partitions, and a new dynamic programming technique for trees, Mathematics of Operations Research 8 (1) (1983) 1–14.

[16] F. Della Croce, U. Pferschy, R. Scatamacchia, New exact approaches and approximation results for the penalized knapsack problem, to appear in Discrete Applied Mathematics.

[17] T. Ibaraki, Y. Nakamura, A dynamic programming method for

single machine scheduling, European Journal of Operational Research 76 (1994) 72–82. doi:10.1016/0377-2217(94)90007-8.
[18] P. Wentges, Weighted dantzig-wolfe decomposition for linear mixed-integer programming, International Transactions in Operational Research 4 (2) (1997) 151–162. doi:10.1111/j.1475-3995.1997.tb00071.x.
[19] A. Pessoa, R. Sadykov, E. Uchoa, F. Vanderbeck, Automation and combination of linear-programming based stabilization techniques in column generation, INFORMS Journal on Computingdoi:10.1287/ijoc.2017.0784.
[20] Ibm ilog cplex optimization studio cplex.
URL www.cplex.com
[21] F. Vanderbeck, Bapcod – a branch-and-price generic code, Tech. rep., University of Bordeaux, INRIA Research team ReAlOpt.

Instance	base			w/o dom			w/o sort			w/o stCB			w/o sgCB			w/o filt			all		
	time	labels/cg	solved	time	labels/cg	solved	time	labels/cg	solved	time	labels/cg	solved	time	labels/cg	solved	time	labels/cg	solved	time	labels/cg	solved
exeo/031	≈ 0	377	1/1	≈ 0	65	1/1	≈ 0	71	1/1	≈ 0	79	1/1	≈ 0	68	1/1	≈ 0	172	1/1	≈ 0	**64**	1/1
exeo/032	≈ 0	1 008	1/1	1	74	1/1	1	**64**	1/1	1	143	1/1	1	83	1/1	≈ 0	259	1/1	1	71	1/1
exeo/054	1	1 999	1/1	≈ 0	141	1/1	≈ 0	**110**	1/1	≈ 0	171	1/1	≈ 0	199	1/1	1	476	1/1	≈ 0	132	1/1
exeo/075	700	2 957	1/1	115	282	1/1	123	277	1/1	108	309	1/1	120	396	1/1	139	554	1/1	111	**252**	1/1
exeo/087	1 224	26 588	1/1	87	2 542	1/1	103	2 447	1/1	100	3 848	1/1	89	2 938	1/1	139	4 464	1/1	80	**2 272**	1/1
exeo/102	2 472	22 875	1/1	162	2 020	1/1	157	1 754	1/1	161	2 343	1/1	175	2 757	1/1	231	3 564	1/1	148	**1 674**	1/1
exeo/109	7 200	61 037	0/1	1 588	2 363	1/1	1 708	2 428	1/1	1 749	4 480	1/1	1 490	2 637	1/1	3 821	8 962	1/1	1 192	**2 090**	1/1
exeo/125	7 200	31 105	0/1	7 200	1 686	0/1	7 200	**1 235**	0/1	7 200	3 068	0/1	7 200	1 944	0/1	7 200	5 386	0/1	7 200	1 559	0/1
exeo/162	2 303	38 191	1/1	353	1 894	1/1	472	**1 332**	1/1	481	5 024	1/1	317	2 136	1/1	514	7 005	1/1	358	1 687	1/1
exeo/163	7 200	85 964	0/1	7 200	5 128	0/1	7 200	4 411	0/1	7 200	8 095	0/1	7 200	7 174	0/1	7 200	8 787	0/1	7 200	**4 383**	0/1
exeo/171	7 200	105 997	0/1	7 200	5 645	0/1	7 200	4 865	0/1	7 200	6 957	0/1	7 200	6 075	0/1	7 200	28 573	0/1	7 200	**4 752**	0/1
exeo/174	7 200	179 300	0/1	1 247	14 559	1/1	1 346	13 140	1/1	1 246	16 682	1/1	1 019	14 173	1/1	3 719	35 935	1/1	992	**11 363**	1/1
exeo/245	7 200	291 000	0/1	7 200	176 685	0/1	7 200	**161 482**	0/1	7 200	206 605	0/1	7 200	216 524	0/1	7 200	258 996	0/1	7 200	204 946	0/1
	time	labels/cg	solved	time	labels/cg	solved	time	labels/cg	solved	time	labels/cg	solved	time	labels/cg	solved	time	labels/cg	solved	time	labels/cg	solved
cvrp/att48	3 725	2 674	**17/25**	3 922	448	14/25	3 943	330	**17/25**	3 887	595	16/25	3 918	441	15/25	3 865	723	**17/25**	3 855	375	**17/25**
cvrp/eil7	≈ 0	17	25/25	≈ 0	5	25/25	≈ 0	5	25/25	≈ 0	7	25/25	≈ 0	5	25/25	≈ 0	9	25/25	≈ 0	5	25/25
cvrp/eil13	≈ 0	67	25/25	≈ 0	31	25/25	≈ 0	28	25/25	≈ 0	47	25/25	≈ 0	32	25/25	≈ 0	38	25/25	≈ 0	30	25/25
cvrp/eil22	≈ 0	624	25/25	≈ 0	225	25/25	≈ 0	163	25/25	≈ 0	488	25/25	≈ 0	229	25/25	≈ 0	244	25/25	≈ 0	213	25/25
cvrp/eil23	4	754	25/25	3	315	25/25	2	214	25/25	2	319	25/25	3	346	25/25	4	585	25/25	2	288	25/25
cvrp/eil30	9	3 225	25/25	2	901	25/25	2	650	25/25	4	1 515	25/25	2	1 028	25/25	2	984	25/25	2	815	25/25
cvrp/eil31	1	756	25/25	1	276	25/25	1	257	25/25	1	378	25/25	1	283	25/25	1	336	25/25	1	264	25/25
cvrp/eil33	177	5 687	25/25	79	3 061	25/25	41	1 978	25/25	84	4 019	25/25	66	3 155	25/25	58	2 848	25/25	51	2 590	25/25
cvrp/eil51	5 328	31 476	12/25	568	1 957	24/25	475	1 346	**25/25**	904	4 765	24/25	598	2 448	24/25	821	4 138	24/25	520	1 790	24/25
cvrp/eilA101	7 200	283 215	0/25	7 200	27 637	0/25	7 200	22 163	0/25	7 200	69 101	0/25	7 200	37 238	0/25	7 200	71 458	0/25	7 200	24 622	0/25
cvrp/eilA76	7 200	25 342	0/25	6 358	1 119	4/25	6 164	947	**7/25**	6 458	2 717	4/25	6 374	1 341	4/25	6 366	2 651	4/25	6 379	1 049	4/25
cvrp/eilB101	7 200	50 507	0/25	7 031	2 255	**2/25**	6 992	2 275	1/25	7 071	5 655	1/25	6 977	2 673	**2/25**	7 039	7 531	1/25	7 038	2 091	**2/25**
cvrp/eilB76	6 529	4 451	3/25	6 446	513	3/25	6 612	455	3/25	6 585	908	3/25	6 441	561	3/25	6 233	1 171	**4/25**	6 436	493	3/25
cvrp/eilC76	7 081	62 620	1/25	5 707	2 028	7/25	5 455	1 298	**8/25**	6 141	6 833	7/25	5 761	2 455	7/25	5 911	6 080	7/25	5 665	1 880	7/25
cvrp/eilD76	7 200	130 733	0/25	6 540	5 161	6/25	6 244	3 484	**7/25**	6 970	17 314	2/25	6 672	6 940	5/25	6 971	14 008	2/25	6 290	4 538	**7/25**
cvrp/gil262	7 200	536 548	0/25	7 200	198 712	0/25	7 200	183 170	0/25	7 200	252 277	0/25	7 200	196 805	0/25	7 200	237 222	0/25	7 200	187 191	0/25
Total (/400)			**208**			**235**			**243**			**232**			**235**			**234**			**239**

Table 2: Performances of the dynamic programming oracle

Instance	BB time	BB t_MP	BB solved	DP1 time	DP1 t_MP	DP1 solved	DP2 time	DP2 t_MP	DP2 solved
exeo/031	≈ 0	≈ 0	1/1	≈ 0	≈ 0	1/1	≈ 0	≈ 0	1/1
exeo/032	≈ 0	≈ 0	1/1	1	≈ 0	1/1	1	≈ 0	1/1
exeo/054	≈ 0	≈ 0	1/1	≈ 0	≈ 0	1/1	≈ 0	≈ 0	1/1
exeo/075	65	11	1/1	111	23	1/1	123	23	1/1
exeo/087	26	1	1/1	80	4	1/1	103	4	1/1
exeo/102	48	8	1/1	148	15	1/1	157	13	1/1
exeo/109	765	20	1/1	1 192	44	1/1	1 708	45	1/1
exeo/125	7 200	128	0/1	7 200	238	0/1	7 200	154	0/1
exeo/162	178	2	1/1	358	7	1/1	472	7	1/1
exeo/163	7 200	1 305	0/1	7 200	551	0/1	7 200	476	0/1
exeo/171	7 200	127	0/1	7 200	126	0/1	7 200	101	0/1
exeo/174	635	26	1/1	992	25	1/1	1 346	28	1/1
exeo/245	7 200	145	0/1	7 200	4	0/1	7 200	5	0/1

Instance	BB time	BB t_MP	BB solved	DP1 time	DP1 t_MP	DP1 solved	DP2 time	DP2 t_MP	DP2 solved
cvrp/att48	3 182	795	16/25	3 855	404	**17/25**	3 943	357	**17/25**
cvrp/eil7	≈ 0	≈ 0	25/25	≈ 0	≈ 0	25/25	≈ 0	≈ 0	25/25
cvrp/eil13	≈ 0	≈ 0	25/25	≈ 0	≈ 0	25/25	≈ 0	≈ 0	25/25
cvrp/eil22	≈ 0	≈ 0	25/25	≈ 0	≈ 0	25/25	≈ 0	≈ 0	25/25
cvrp/eil23	3	≈ 0	25/25	2	≈ 0	25/25	2	≈ 0	25/25
cvrp/eil30	1	≈ 0	25/25	2	≈ 0	25/25	2	≈ 0	25/25
cvrp/eil31	2	≈ 0	25/25	1	≈ 0	25/25	1	≈ 0	25/25
cvrp/eil33	12	1	25/25	51	1	25/25	41	1	25/25
cvrp/eil51	212	25	**25/25**	520	18	24/25	475	16	**25/25**
cvrp/eilA101	6 960	157	**1/25**	7 200	13	0/25	7 200	16	0/25
cvrp/eilA76	5 385	513	**12/25**	6 376	164	4/25	6 164	148	7/25
cvrp/eilB101	6 716	323	**2/25**	7 038	129	**2/25**	6 991	109	1/25
cvrp/eilB76	5 910	429	**9/25**	6 436	181	3/25	6 612	162	3/25
cvrp/eilC76	4 651	425	**11/25**	5 665	148	7/25	5 455	155	8/25
cvrp/eilD76	4 762	531	**13/25**	6 290	104	7/25	6 244	115	7/25
cvrp/gil262	7 200	61	0/25	7 200	5	0/25	7 200	5	0/25
Total (/400)			264			239			243

Table 3: Comparison of the branch-and-bound oracle and dynamic programming oracle

Instance	time_tot			time_root			gap_opt = (pb-db)/db			gap_root = (pb*-lp)/lp			nb_bb_nodes			time_per_node		
	(P_0)	(P)	CG BB	(P_0)	(P)	CG BB	(P_0)	(P)	CG BB	(P_0)	(P)	CG BB	(P_0)	(P)	CG BB	(P_0)	(P)	CG BB
exeo/031	45	25	**≈ 0**	**≈ 0**	**≈ 0**	≈ 0	0.0%	0.0%	0.0%	242.6%	87.8%	**0.0%**	2574	1 759	1	0	**0**	0
exeo/032	21	3	**≈ 0**	**≈ 0**	≈ 0	≈ 0	0.0%	0.0%	0.0%	143.5%	26.3%	**3.9%**	931	215	9	**0**	**0**	0
exeo/054	393	6	**≈ 0**	≈ 0	1	**≈ 0**	0.0%	0.0%	0.0%	126.3%	5.2%	**0.0%**	2885	38	1	**0**	0	0
exeo/075	-	260	**65**	2	**1**	2	29.0%	0.0%	0.0%	128.6%	25.1%	**7.4%**	5659	1 054	145	1	**0**	0
exeo/087	-	2 981	**26**	5	**4**	5	47.7%	0.0%	0.0%	171.0%	66.5%	**2.9%**	19882	5 023	21	**0**	1	1
exeo/102	-	4 126	**48**	8	12	**5**	57.0%	0.0%	0.0%	154.8%	39.5%	**3.3%**	1212	2 613	21	6	**2**	2
exeo/109	-	-	**765**	**12**	19	18	214.4%	82.2%	**0.0%**	271.4%	101.8%	**3.5%**	1672	5 983	157	4	**1**	5
exeo/125	-	-	-	**27**	30	32	∞	126.3%	**0.5%**	293.7%	120.9%	**8.0%**	1367	3 019	1 677	5	**2**	4
exeo/162	-	-	**178**	71	178	**59**	∞	143.1%	**0.0%**	319.2%	67.0%	**0.9%**	429	520	17	17	14	**10**
exeo/163	-	-	-	83	223	**39**	218.9%	60.4%	**12.6%**	160.5%	50.8%	**12.8%**	470	541	309	15	**13**	23
exeo/171	-	-	-	**98**	300	156	∞	∞	**3.5%**	310.6%	69.1%	**5.6%**	310	181	288	**23**	40	25
exeo/174	-	-	**635**	**97**	275	205	∞	∞	**0.0%**	251.8%	74.6%	**1.1%**	107	433	21	67	**17**	30
exeo/245	-	-	-	**306**	1 233	576	72.4%	**14.8%**	67.5%	140.3%	26.6%	**2.9%**	151	17	4	**48**	424	2 788

Instance	(P_0)	(P)	CG BB
cvrp/att48	5/25	16/25	16/25
cvrp/eil7	25/25	25/25	25/25
cvrp/eil13	25/25	25/25	25/25
cvrp/eil22	25/25	25/25	25/25
cvrp/eil23	25/25	25/25	25/25
cvrp/eil30	25/25	25/25	25/25
cvrp/eil31	25/25	25/25	25/25
cvrp/eil33	25/25	25/25	25/25
cvrp/eil51	0/25	0/25	25/25
cvrp/eilA101	0/25	0/25	1/25
cvrp/eilA76	0/25	0/25	12/25
cvrp/eilB101	0/25	0/25	2/25
cvrp/eilB76	0/25	0/25	9/25
cvrp/eilC76	0/25	0/25	11/25
cvrp/eilD76	0/25	0/25	13/25
cvrp/gil262	0/25	0/25	0/25
Total	**180/400**	**191/400**	**264/400**

Table 4: Performances of the extended formulation

Appendix A. Comparison of the branch-and-bound oracle and dynamic programming oracle

In this appendix, we report the results of additional experiments comparing both pricing oracles. We separated these results from the main document since they are consistent with the ones we presented in subsection 5.4. In these experiments, we observed that the dynamic program can slightly outperforms the branch-and-bound oracle (see `cvrp/eilB76` and `cvrp/eilB101`). In other cases, the oracles' performances are comparable, depending on what metric we use as a reference (see: `exeo/174`, `cvrp/A76` and `cvrp/C76` ; or `exeo/75`, `exeo/102` and `cvrp/gil262`). In all remaining cases, the branch-and-bound algorithm clearly outperforms the dynamic programming approach (see: `cvrp/D76` and `exeo/109` ; or `exeo/162` and `exeo/87` ; or `cvrp/att48` and `cvrp/A101`).

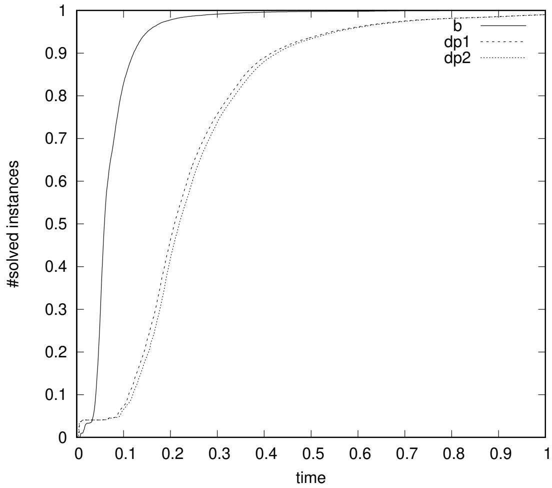


Figure A.15: exeo/087 walltimes

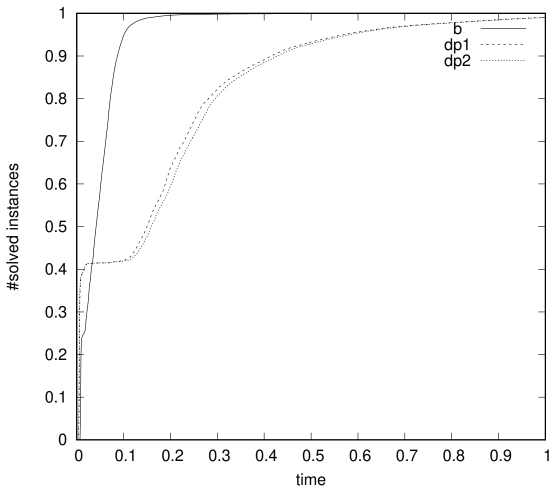


Figure A.16: exeo/102 walltimes

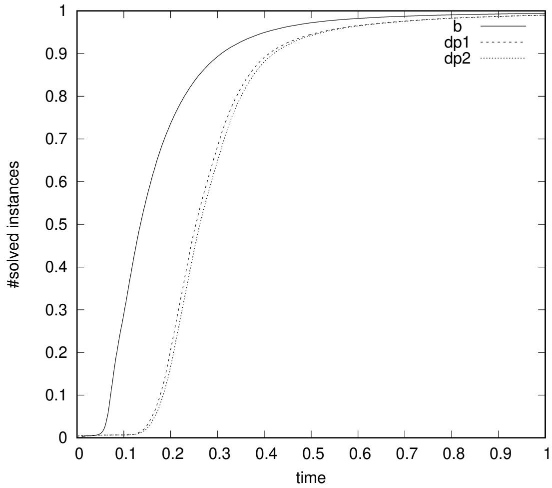


Figure A.17: exeo/109 walltimes

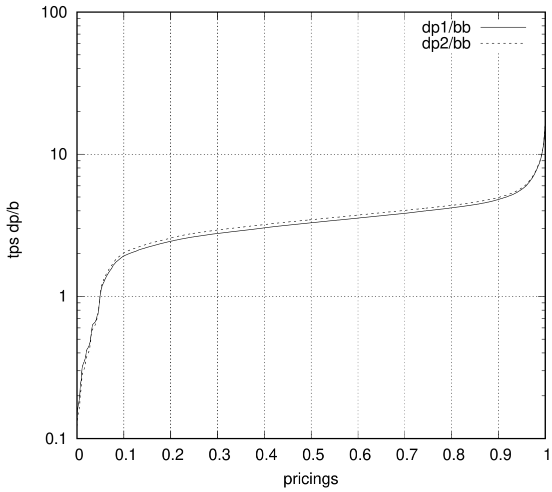


Figure A.18: exeo/087 time ratios

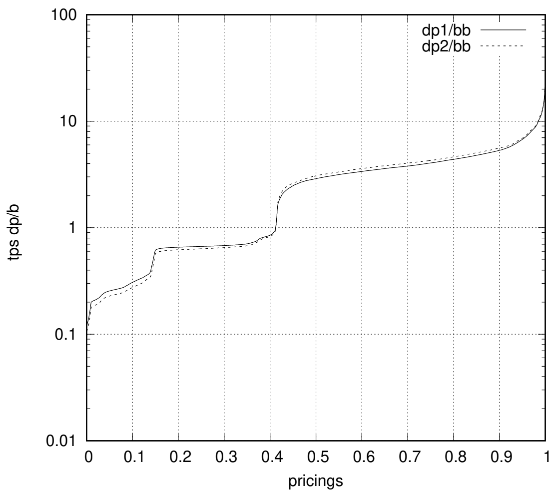


Figure A.19: exeo/102 time ratios

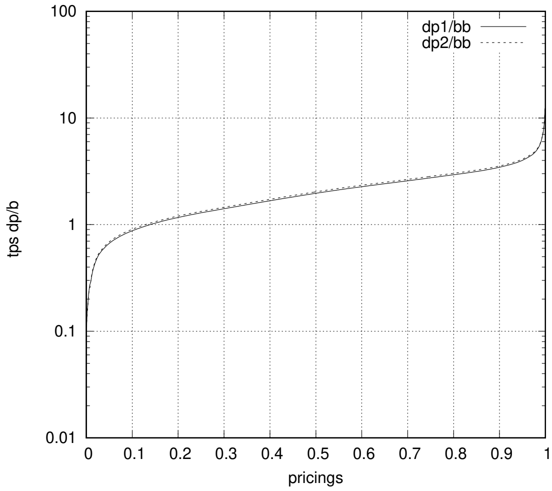


Figure A.20: exeo/109 time ratios

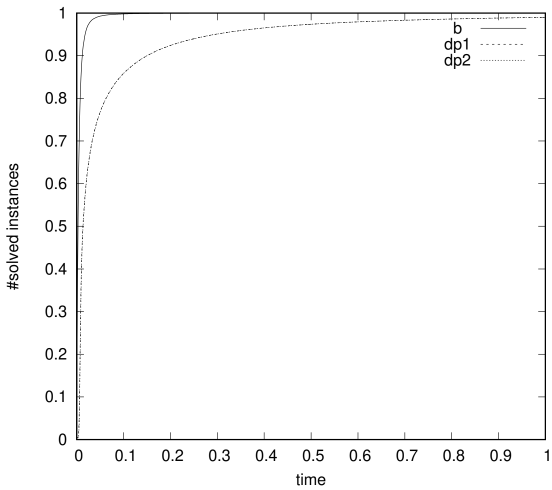


Figure A.21: cvrp/eilA101 walltimes

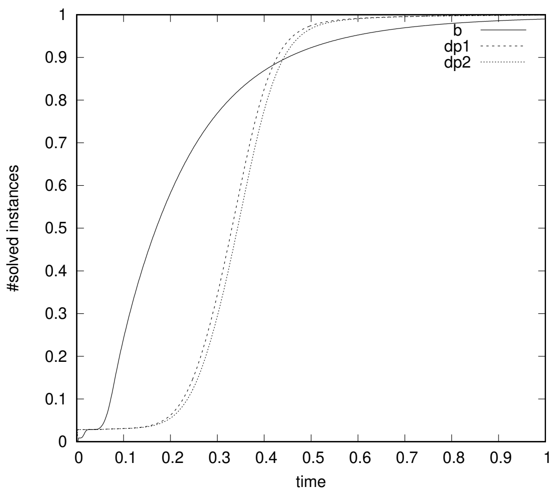


Figure A.22: cvrp/eilA76 walltimes

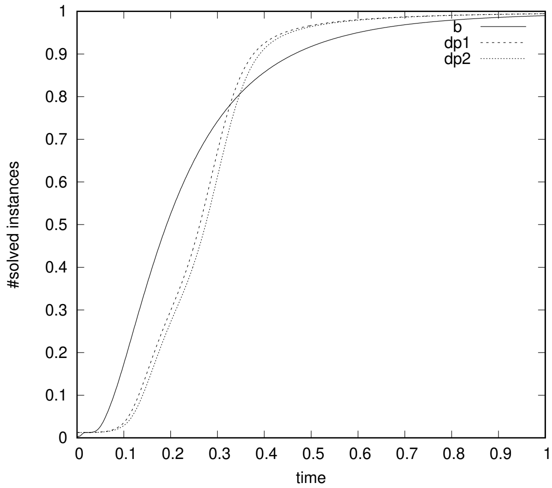


Figure A.23: cvrp/eilB101 walltimes

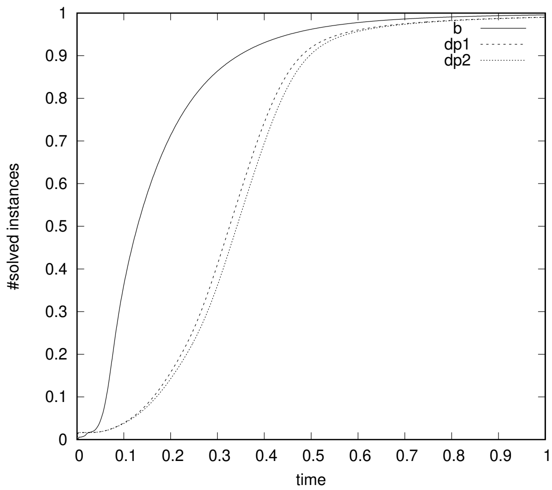


Figure A.24: cvrp/eilC76 walltimes

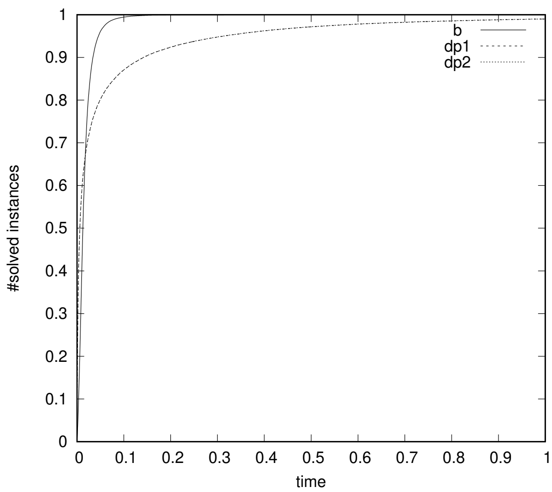


Figure A.25: cvrp/gil262 walltimes

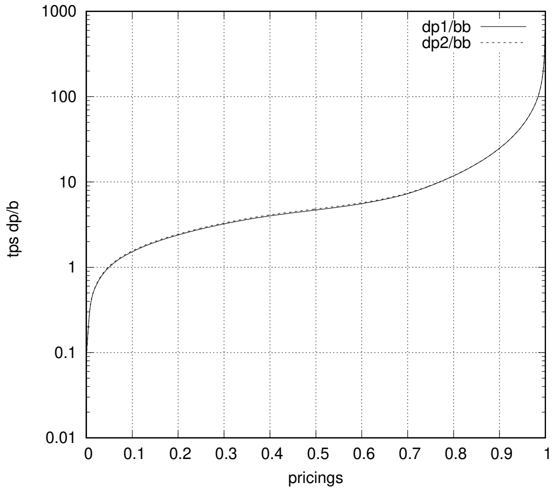


Figure A.26: cvrp/eilA101 time ratios

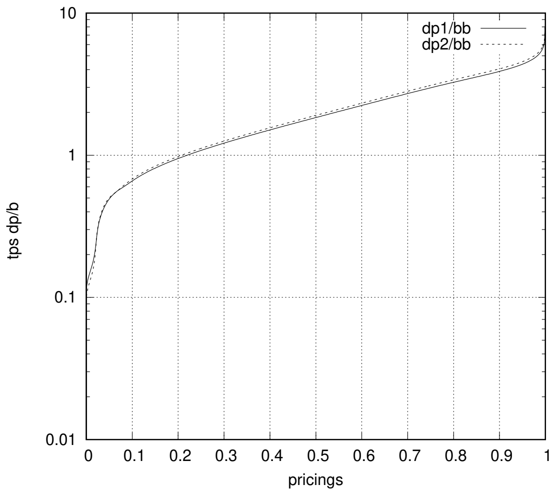


Figure A.27: cvrp/eilA76 time ratios

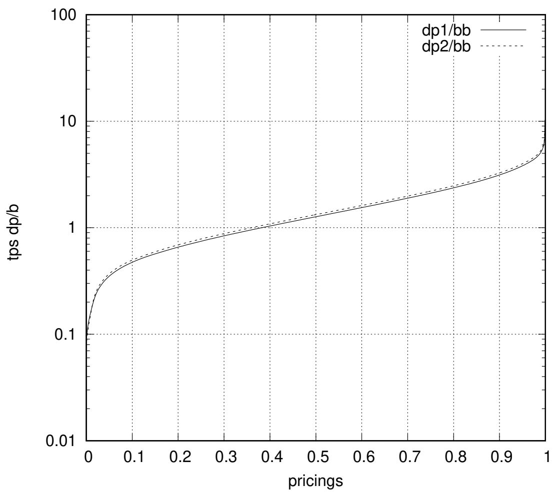


Figure A.28: cvrp/eilB101 time ratios

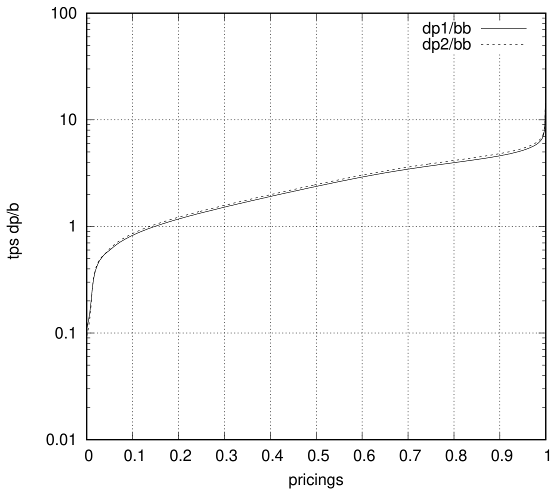


Figure A.29: cvrp/eilC76 time ratios

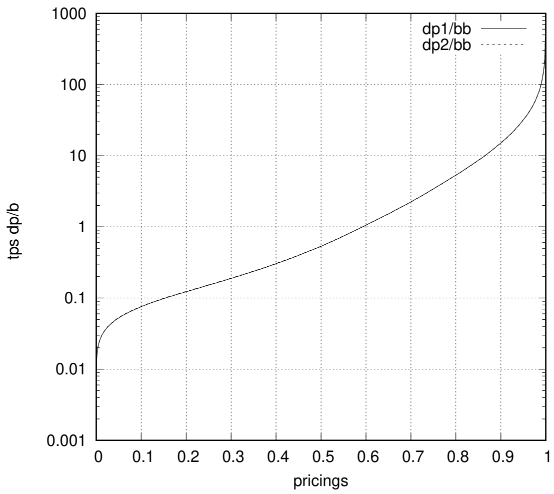


Figure A.30: cvrp/gil262 time ratios